AFIT/GCS/ENG/99M-02

A CONSTRUCTIVE INDUCTION APPROACH TO COMPUTER IMMUNOLOGY

THESIS

Kelley J. Cardinale
Captain, USAF

Hugh M. O'Donnell
Second Lieutenant, USAF

AFIT/GCS/ENG/99M-02

The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

AFIT/GCS/ENG/99M-02

A CONSTRUCTIVE INDUCTION APPROACH TO COMPUTER IMMUNOLOGY

THESIS

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology
Air University
Air Education and Training Command
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Kelley J. Cardinale, B.S.
Captain, USAF

Hugh M. O'Donnell, B.S.
Second Lieutenant, USAF

March 1999

Approved for public release; distribution unlimited

AFIT/GCS/ENG/99M-02

A CONSTRUCTIVE INDUCTION APPROACH TO COMPUTER IMMUNOLOGY

Kelley J. Cardinale, B.S.
Captain, USAF

Hugh M. O'Donnell, B.S.
Second Lieutenant, USAF

Approved:

Greg. H Gunsch
Lt Col Gregg H. Gunsch

8 MAR 99
Date

Gary B Lamont
Dr. Gary B. Lamont

8 Mar 99
Date

Stuart Kramer
Lt Col Stuart C. Kramer

8 MAR 99
Date

John S. Crown
Maj John S. Crown

8 Mar 99
Date

Dedication

To those who kept us on our path to completion,

and to those who knew when to divert us.

Acknowledgments

We would like to express our appreciation to our thesis advisor, Lt Col Gregg Gunsch, for pushing us out of our "comfort zones" during the thesis process. He allowed us the independence to make our own mistakes, as well as experience our own triumphs. We would also like to thank our committee members, Dr. Gary Lamont, Lt Col Stuart Kramer, and Maj John Crown, for their contributions to this endeavor. We are also grateful to Dave Doak, for providing us office space, disk space and much needed fish advice. We would especially like to thank our family and friends for their constant motivation.

I would like to thank my parents for a lifetime of support and encouragement, especially over the last eighteen months. Dad, your past AFIT experiences and humorous thesis stories always gave me perspective. Don't forget, the registrar is still waiting for your GRE scores. Mom, thank you for always seeing the light at the end of the tunnel, even when I didn't. I can finally say, "Yes, Mom, I am finished with my thesis."

Kelley

I would like to thank my parents for the sacrifices they made; they are my inspiration. I would also like to thank Erin for ensuring I paid attention to schedules and other trivialities. I never did mind much about the little things. I also thank my brothers, Danny, Joe and James, for ensuring that I never strayed too far from sanity and humility. Finally, this thesis effort would not have been possible without the contributions of Arthur T. Guinness, Linus Torvalds and Gary Fisher.

Hugh

Table Of Contents

List of Tables

List of Figures

AFIT/GCS/ENG/99M-02

Abstract

With the increasing birth rate of new viruses and the rise in interconnectivity and interoperability among computers, the burden of detecting and destroying computer viruses is severe. This research integrated four different domains: computer virus detection, human immunology, computer immunology and constructive induction. First, a Computer Health System, based on the public health system, was defined that could possibly improve the current "global" approach to computer virus protection. Second, a computer immune model, based on the human immune system, was defined that could possibly improve the current "local" approach to virus detection. Third, the detection component of this computer immune model was developed, represented by the prototype MERCURY. This model utilized the machine learning concept of constructive induction to capture the human immune characteristics of detection, self-adaptation and memory.

The results of analyzing MERCURY demonstrate a lack of representational power of computer virus byte patterns using selective induction. Therefore, constructive induction is needed to provide new, potentially powerful, and often necessary representations. However, the results confirmed constructive induction's main deficiency, the explosion in the number of hypotheses generated. The effects of this deficiency can be improved by utilizing key pieces of knowledge to guide construction. Process optimization through statistical techniques, provides direct insight into these key pieces of knowledge. Knowledge about the virus domain, such as characteristics of typical viruses and regularities in the byte patterns, also provides guidance for effective construction.

A CONSTRUCTIVE INDUCTION APPROACH TO COMPUTER IMMUNOLOGY

1. Introduction

1.1. Motivation

The vision for the 21st Century Air Force includes aggressively expanding the efforts in information warfare, both offensively and defensively. The top priority is to defend capabilities in this arena by continuing to build up the defense of computer systems and improve our tactical and operational information warfare capabilities. With the Air Force's increasing rate of computer dependency, the threat of computer viruses is a rapidly growing concern.

Computer viruses have been a growing concern since the early 1980s [FHS97]. To combat this problem, various antivirus programs have been created; however, the burden of detecting and destroying viruses is still severe. Two trends are inhibiting the effectiveness of current antiviral techniques: the increasing birth rate of new viruses and the rise in interconnectivity and interoperability among computers [Kep94]. Current techniques are reactive, labor intensive for virus researchers, have a slow response from time of discovery until the cure is prescribed, and require user intervention to update the virus signature database. Improving current antiviral techniques can combat these problems [Kep94].

Based on several properties of the human immune system, computer scientists hope new techniques to fight computer viruses will emerge. They believe that the human immune system has several useful characteristics for detecting computer viruses. These properties provide a robust, flexible and scalable system resilient to attack and a fresh perspective on computer viruses and other security problems [FHS97].

Human immune systems are unique, meaning *individual* immunity is derived and adapted differently; this is a desirable and applicable property of a computer system, as well. The immune system uses a decentralized and distributed detection process, also of interest in the virus detection domain. The human system is very flexible and does not require the *absolute* detection of every invader; instead, *partial* detection allows for quicker recognition of multiple invaders. In a computer system, this is similar to the use of byte pattern signatures as partial detectors for locating an infected file on a computer system.

Another very important feature of the immune system is its ability to detect and react to invaders, or "nonself," while not inappropriately detecting what belongs in the body, or "self." This property applies to invaders that have been previously seen, as well as those previously unseen. The human system can learn the structures of these previously unseen invaders and remember them, so that the body's future responses to the same invader can be faster.

These last three human immune properties of detection, adaptation, and memory are the most important throughout this research. These strengths of the human immune system are the *foundation* for the development of a new antivirus detection method, investigated in this research.

Drawing on some of the strengths of the human system, other researchers have proposed computer immune models. These computer immune models incorporate many of the positive aspects of current antiviral techniques while introducing improvements such as automatic distribution of viral prescriptions, automatic recognition and removal of viruses, and protection of self [MVL98]. These models each provide an overall approach to an *aspect* of computer immunology, but do not always give a full picture that includes virus detection on an individual system.

The research conducted in our investigation presents two *computer models*. The first model provides a "global approach" to computer health by incorporating individual computer systems through a computer health infrastructure. The second model provides a "local approach" to computer health by utilizing a method of virus detection, based on the properties of the human immune system.

To properly model this individual detection system after the human immune system, a learning mechanism is required for the adaptation property. This mechanism provides an automated method for distinguishing between positive and negative instances of self that defines the overall *concept* of self. This learning mechanism is adaptive since it allows the definition of self to change; it "learns" by incorporating new viruses, representing nonself, into this definition.

The specific learning mechanism utilized in this research is based on a form of machine learning called constructive induction. This mechanism derives knowledge from the observation of positive and negative examples of a concept, in this case, self. This type of learning attempts to characterize the concept by creating a description that captures the essence of the concept while distinguishing it from the counterexamples. This is accomplished by carefully selecting from the original attributes describing the examples and, when necessary, constructs new, more useful, attributes. This form of learning provides a mechanism for learning meaningful descriptions of complex data, such as distinguishing between self, those components authorized as part of the system, and nonself that does not belong on a computer system.

Analogies between the human immune system and the computer immune system can be derived through constructive induction. Both systems create potential detectors for identifying invaders. There are differences, though. The body can afford to retain billions of detectors for "future use," and the inherently parallel nature of cellular activity allows constant testing against substances without performance degradation. The computer cannot maintain and test a large number of detectors without a loss in system performance. To create a practically useful system, complex detectors must be induced through the selection of byte patterns and the application of operators defining their interrelationships. Unsuitable or repetitive detectors are filtered out through testing.

Other machine learning approaches, such as evolutionary algorithms and neural networks, also entail the necessary properties of construction and induction. Since they can be shown to perform "limited" construction based upon their sets of operators, this research focuses on the more *general* and *deterministic* learning program known as constructive induction.

1.2. Problem

Due to the lack of an adaptation mechanism, many virus detection methods are insufficient at the individual system level. In addition, the global approach to computer health is inadequate. Though several immunological approaches have been studied, many lack a mechanism for the adaptive detection of viruses on an individual computer system. This research proposed utilizing the machine learning mechanism of constructive induction for the adaptive detection. However, the most difficult issue faced by constructive induction, as a field of machine learning, is the selection of the most useful operator-attribute combinations. These combinations create a computational explosion, necessitating the use of *a priori* knowledge to make the learning process more efficient.

1.3. Research Objectives

This research integrated four different domains, depicted in Figure 1: computer virus detection, human immunology, computer immunology and constructive induction. First, a Computer Health System, based on the public health system, was defined that could possibly improve the current "global" approach to computer virus protection. Second, a computer immune model, based on the human immune system, was defined that could possibly improve the current "local" approach to virus detection. Third, the detection component of this computer immune model was developed, represented by the prototype MERCURY. This model utilized the machine learning concept of constructive induction to capture the human immune characteristics of detection, self-adaptation and memory.

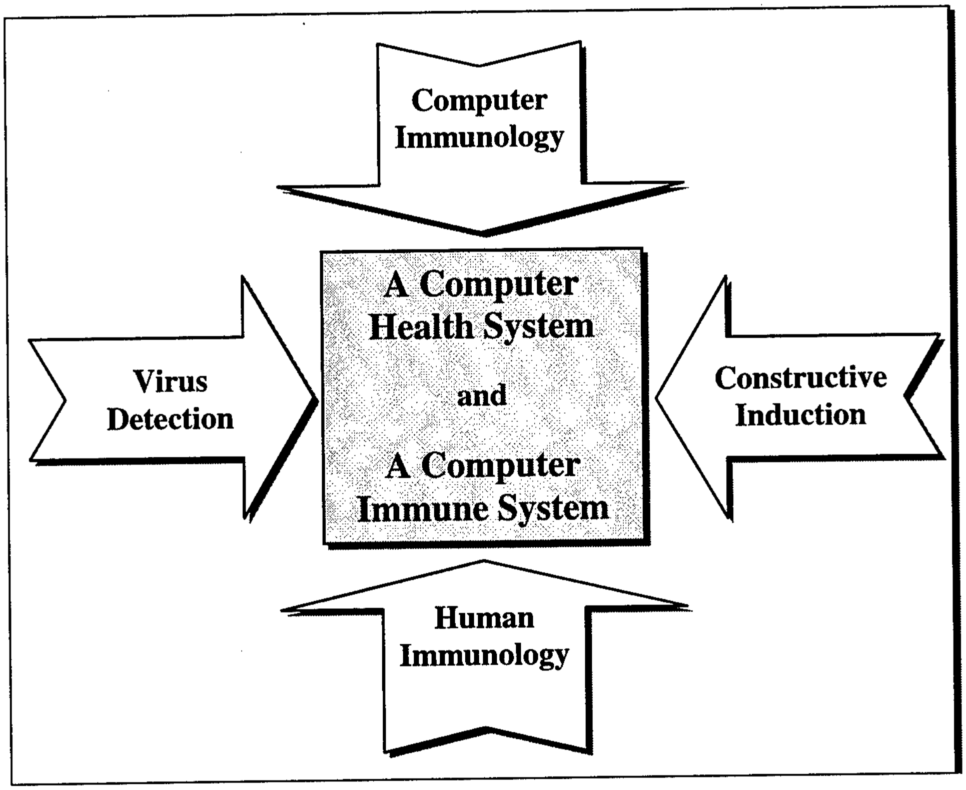

Figure 1 -- Research Domains

The work accomplished as part of this investigation tested the following primary hypotheses:

Primary Hypotheses

1. The public health system is a useful model for a Computer Health System for the global protection of computer system against viruses
2. The human immune system is a useful model for a virus detection system on an individual computer system
3. Constructive induction provides a suitable learning mechanism for the virus detector system of an individual computer system

The first two objectives of this research were to test the first two hypotheses to determine if the public health system and the human immune system were useful models for a Computer Health System and computer immune system, respectively. To accomplish this objective, research in the areas of public health and human immunology was conducted. The requirements, objectives and components of the models were also evaluated. Both computer models are informal, explanatory models based on some essential qualities of their respective systems. Due to the models' informalities, though, not all of their aspects were explicitly stated.

The third objective of this research was to empirically test the third hypothesis. This objective investigated whether constructive induction was suitable for virus detection in a computer immune system. Testing was conducted utilizing MERCURY. While MERCURY captures the essence of constructive induction, it does not fully employ all the characteristics of a complete inductive engine. Therefore, based on the empirical analysis of MERCURY's results, the hypothesis that constructive induction provides a suitable learning mechanism for the virus detector system of an individual computer system can not be accepted or rejected. Rather, the results of these tests provided empirical evidence and analytical knowledge that could make the learning process more efficient. Since the main disadvantage of constructive induction is its computational explosion, these results provided mathematically based methods which could decrease its computational complexity. These methods could improve the capabilities of a fully developed constructive induction based virus detector, by providing knowledge about the problem domain and the system parameters *a priori.* To confirm these findings, a process optimization simulation was conducted to demonstrate the effectiveness of *a priori* knowledge applied to the virus detection problem.

The third hypothesis was decomposed into smaller, more manageable, sub-hypotheses. The first sub-hypothesis was the Virus Feature Hypothesis:

Virus Feature Hypothesis

Byte patterns can be used as the basis of a constructive induction based computer virus detector.

This hypothesis captured the belief that byte patterns extracted from computer files are "adequate" features for distinguishing between positive and negative instances of self, defining the overall concept of self, and adapting to a changing definition of self. Current virus knowledge [KeA94] concludes that the use of byte patterns is an effective method for detecting a broad variety of conceivable mutations for a particular virus with a low false positive probability. To confirm this, MERCURY's learning component was programmed to extract, manipulate, and test byte patterns from various files. Testing concluded that the learning component, using features composed of byte patterns, was able to distinguish between self and nonself files with varying degrees of accuracy. Therefore, it can be concluded that byte patterns can be used as the basis of a constructive induction based computer virus detector.

The second sub-hypothesis was the Constructive Operator Hypothesis:

> **Constructive Operator Hypothesis**
>
> Logical and spatial operators can be used for constructing new attributes for the computer virus detector.

This hypothesis validated that the choice of operators was "adequate" to construct new features, better distinguishing between infected and uninfected data. Current virus knowledge [KeA94] confirms the applicability of using relative and absolute locations of virus characteristics in a file, and the presence of multiple virus characteristics throughout one file to detect viruses. To confirm this, MERCURY's learning component was programmed to manipulate the byte patterns from various files based on two types of operators, logical and spatial. The logical operators AND, OR and XOR accounted for multiple, or restricted virus characteristics, whereas the spatial operators BEFORE and DISTANCE accounted for the positions of virus characteristics. Testing concluded that the learning component, using logical and spatial operators, was able to distinguish between self and nonself files with varying degrees of accuracy. Therefore, it can be concluded that logical and spatial operators can be used for constructing new features for the computer virus detector.

The two sub-hypotheses are not independent. A stable definition of self, and the ability to distinguish between self and nonself are dependent on a combination of virus features and constructive operators. The difficulty of this problem is large due to the complexity caused by increasing the number of operators, features or both. The combination of these system parameters results in a large number of constructed attributes; therefore, some limitations were established. Evaluating these two sub-hypotheses separately and together resulted in a more effective basis for supporting the primary hypothesis. More importantly, the results of these sub-hypotheses provide the empirical analysis needed to improve the effectiveness of the overall learning process.

1.4. Approach

This research contained three phases: model building, prototype development, and testing and analyses. The first phase built two computer models, the Computer Health System and the individual computer immune system. This modeling derived, by analogy, the important properties, functions, and requirements for the two computer models from the public health system and human immune system, respectively. The relationship between the two models is depicted in Figure 2.

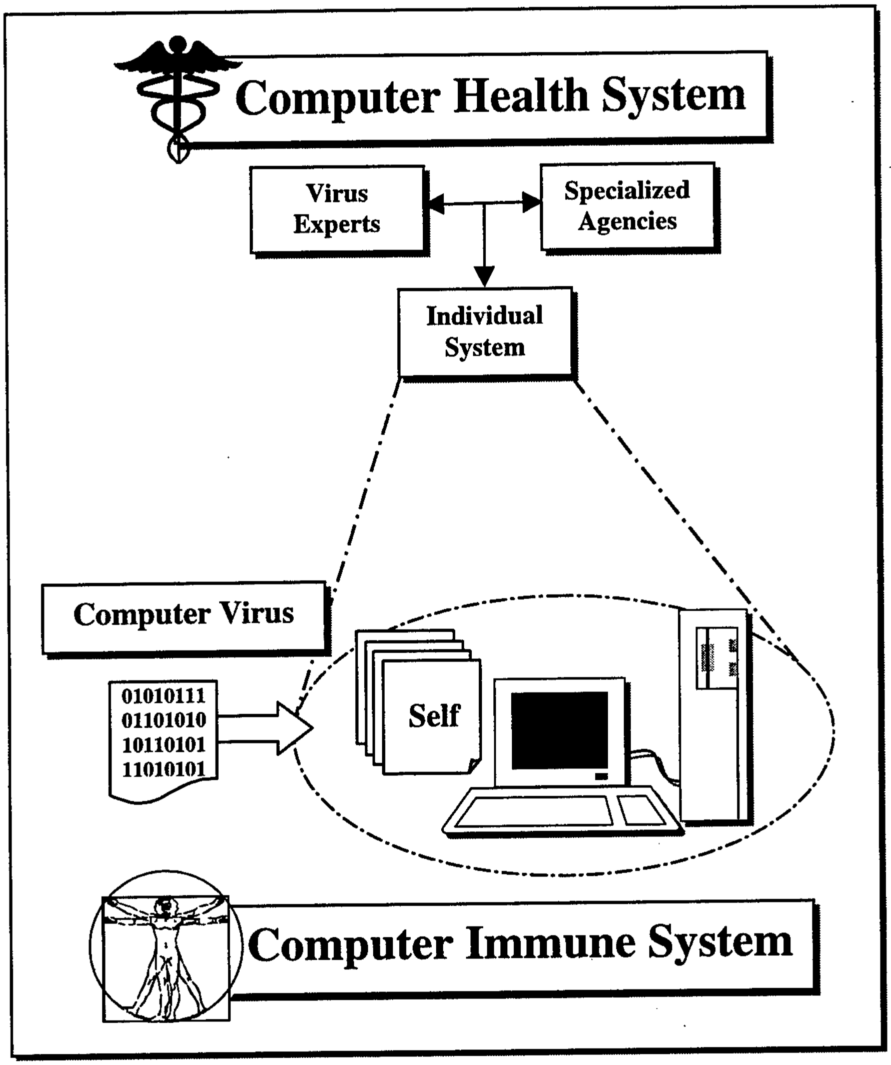

Figure 2 – The Computer Health and Computer Immune Relationship

Once modeling phase was complete, the second phase designed and developed the prototype MERCURY, capturing the essence of the individual computer immune model. MERCURY encompasses the human immune properties of detection, adaptation, and memory, and is comprised of three components: the virus scanner; the learning engine, HEC; and the knowledge base.

The third phase included the testing and analyses of MERCURY. This system was trained on several computer files representative of self and nonself files on a computer. The expected output was a set of byte patterns that could distinguish between self and nonself files. Testing of MERCURY was accomplished through two groups of test cases. The first group tested whether byte patterns, the rules combining the byte patterns, and the constructive operators used to construct new features from these byte patterns were applicable in the constructive induction approach to virus detection. The second group tested whether constructive induction provides a suitable learning mechanism for the virus detector system of an individual computer system. Additionally, the overall results were studied in order to focus this learning method, reducing its computational complexity.

1.5. Scope

There are two methods of infection in the human body, intracellular and extracellular. *Intracellular* infections in the body are similar to abnormal byte patterns in a file since determination of an infection requires the "extraction" of information from a computer's version of a cell, the file. Conversely, *extracellular* infections, which are not attributed to a specific cell, are analogous to the detection of viruses through heuristic tests and analyses of system calls and abnormal system activity. MERCURY focused on the *intracellular* type of infection in the computer immune system, based on the method in which the virus infects the file, and the system's response to the infection.

Instead of focusing efforts on implementing a fully operational virus detection component, the focus of this research was the application of constructive induction to the virus detection domain. This research investigated the design and initial development of the adaptation mechanism based on this from of machine learning. This initial development included the choice of distinguishing features, selective induction rules, and constructive induction operators. Additionally, this research focused on improving the computational complexity of the learning process, in order to increase the applicability of this form of learning.

This research recognized specific machine learning improvements that could be applied to current methods of virus detection, in order to improve performance. It also presented the incorporation of this constructive induction component into an individual computer's immune system, and further incorporated this system into an overall global picture of computer health.

1.6. Thesis Overview

During this investigation, two computer immune models were presented: the Computer Health System and the computer immune system. MERCURY, the virus detection component of an individual computer immune system, was prototyped by integrating a constructive induction engine, HEC, with a virus scanning program, and a knowledge base.

This document provides a discussion of this research effort, beginning in Chapter Two with a literature review of viruses, virus detection methods, constructive induction, and machine learning applications. Chapter Three provides an overview of the human immune system and the public health system, presents previous research in the area of computer immunology, introduces the models of the Computer Health System and the computer immune system, and presents the virus detection component, MERCURY. Chapter Four provides insight into MERCURY's design methodology, followed by a detailed discussion of its design and implementation in Chapter Five. Chapter Six presents the results of the system testing. Finally, Chapter Seven presents the conclusion that a constructive induction approach to computer immunology could improve computer virus detection. Additionally, Chapter Seven presents the conclusion that a computer immune system should be recognized and incorporated as a valuable component in a higher-level Computer Health System. This chapter also provides areas for future research within the various domains of computer immunology, machine learning, and virus detection.

2. Literature Review

2.1. Overview

This chapter presents the background knowledge needed to develop the constructive induction based computer virus detection component, MERCURY. The first section introduces computer viruses by describing their function and structure, and reviewing the different types of viruses known to infect computer systems today. The next section elaborates on the current methods used for virus detection and pinpoints some of their inadequacies. The last sections describe various machine learning methods, specifically the method of constructive induction used by MERCURY, as well as current research trends, which use other types of machine learning in computer security and intrusion detection applications.

Chapter Three provides additional background knowledge in human immunology and public health needed to build the two computer models. It presents the Computer Health System, and the computer immune system of which MERCURY is a component.

2.2. Computer Viruses

Solid technical knowledge is the foundation for all viral defenses [Lud98]. To build an effective virus detector, an understanding of virus structure, function and behavior is needed. The following sections describe computer viruses by classifying the different types commonly encountered today, defining their functions, and examining their different structures and infection methods.

2.2.1. Definition and Structure

A computer virus is a block of executable code that attaches itself to, overwrites, or replaces another program in order to reproduce itself without the knowledge of the user [NCSA96]. Computer viruses replicate by attaching to a host, usually a program or computer, and utilizing the host's resources to make copies of themselves. Computer viruses spread from computer to computer, in the same way that biological viruses spread among individual members of society [KSCW97].

A typical computer virus performs two functions. First, it copies itself into previously uninfected programs or files. Second, it executes the payload, or the intent of the virus. Common viral payload effects include deleting files, modifying files, displaying messages on-screen, or updating programs. Payloads can be damaging, amusing, or possibly even useful, but nevertheless unwanted and uncontrolled. A virus may cause damage by replicating itself and taking up scarce resources, such as disk space, CPU time, or network connections [WCC89]. A virus requires manual intervention to begin its infection, but will continue to automatically infect programs once it is started [Che97].

Every computer virus contains must contain at least two routines, *search* and *copy* [Lud98]. The search routine locates new files or disks as targets for infection, determines which resource to infect and how often infection will occur. The virus' copy routine copies the virus into the resource, located by the search routine. There is a size vs. functionality tradeoff, though; the more sophisticated these routines are, the more space they will take up, easing detection.

In addition to these two routines, some computer viruses contain *anti-detection* routines. These routines vary in complexity, from keeping the date in a file the same upon infection to camouflaging the virus completely. Most viruses may contain *destructive* routines, which carry out the *possibly* malicious "intent" of the virus. [Lud98]

2.2.2. Classification Methods

Viruses can be classified according to the following characteristics: environment, operating system, different algorithms at work, and destructive capabilities. The following table provides a cursory overview of these types.

Table 1 -- Classification of Viruses

Environment	*File Infector Viruses*	*Infect executables in various ways, create file doubles, or use file system specific features.*
	Boot Sector Viruses	*Save themselves in the disk boot sector or to the Master Boot Record, or change the pointer to an active boot sector.*
	Macro Viruses	*Infect document files, spreadsheets and databases of several popular software packages.*
	Network Viruses	*Use protocols and commands of computer network or email to spread themselves.*
Algorithms at Work	*TSR Capability*	*Infects a computer and leaves its resident part in RAM, which interrupts system calls to target objects and incorporates into them.*
	Stealth Algorithms	*Allows viruses to completely or partially cover their traces inside the operating system. The most common use is the interception of operating read/write calls to infected objects; stealth viruses "substitute themselves with uninfected pieces of information.*
	Self Encryption and Polymorphic Capability	*Hard to detect due to absence of signatures; none of their code fragments remain unchanged. This may be achieved by encrypting the main body of the virus and making modifications to the decryption routine.*
	Non-standard Techniques	*Used to hide viruses deep within the operating system kernel; protects against detection and more difficult to remove.*
Destructive Capabilities	*Harmless*	*Have no effect on computing, except for a lowering of free space because of propagation and reduction in CPU utilization.*
	Not Dangerous	*Limit their effect to lowering free disk space and a few graphical, sound or other functions.*
	Dangerous	*Can seriously disrupt the computer's work.*
	Very Dangerous	*Contain routines that may lead to the loss of data, data destruction, and erasure of vital information in system areas.*

2.2.3. Types

Further classification of computer viruses can be accomplished according to the types of programs they infect and the method of infection employed [Lud98]. One such classification is between boot sector infectors and file infectors. Other viral types include macro viruses, stealth viruses, and polymorphic viruses. The following sections describe some of these common viruses. The file infector viruses are explained in detail, since they are the type of virus MERCURY was developed to detect. This is information is needed for determining the best methods for extracting byte patterns from the files, the best operators for combining the byte patterns, and the overall knowledge needed for applying constructive induction to this domain.

2.2.3.1. File Infector

File infector viruses affect the program files that a system must load in order to make software function. When the program executes, the virus code executes and infects more files [Lud98]. According to this method of infecting files, viruses are divided into overwriting, parasitic, companion and link viruses [Kas99].

The *overwriting* method of infection is the simplest; the virus overwrites the contents of a target executable with its own code, destroying the original contents of the target file. The executable file stops working properly and can not be restored.

Parasitic viruses are file viruses that change the byte ordering within target files while transferring copies of themselves, but the files themselves remain at least partially usable. These parasitic viruses can be "prepending," by saving themselves at the top of file; "appending," by saving themselves at the end of file; "inserting," by inserting themselves in the middle of file; or "cavity," by copying of its own code to such parts of the file which are known to be unused. [Kas99]

Virus incorporation at the top of a file is the most widely used method of insertion into DOS *.bat ,*.com and *.exe files [Kas99]. There are two known methods of inserting a parasitic file virus at the top of file. The first involves copying the top of the target file to the end of file and then copying the virus body to the free space at the top of file. In the second method the virus creates a copy of itself in RAM, appends the target file and then saves the resulting concatenation to disk.

Another common method of virus incorporation into a file is appending the virus to the end of file. In this method, the virus must also change the top of file so that the virus code is executed first. In DOS *.com files, this is achieved by changing the first several bytes to the instruction codes or to the address of the routine passing control to the body of virus.

The final method of insertion into a file involves incorporating the virus into the middle of the file. In the simplest case the virus "spreads" the file by moving fragments of the file to the end and then writes its own code into the free space. Some viruses will even compress their inserted fragments so that the file size remains unchanged. A more difficult method is called "cavity" insertion, where a virus copies itself to unused areas of the file.

The last two file infector viruses are companion and link. *Companion viruses* are another type of file infector virus that do not change the infected files. They operate by creating a copy of the target file and replacing the original with itself. When the target file executes, the virus gets the control instead of the original file. *Link viruses*, like companion viruses, do not change the physical contents of files. However, when an infected file is executed, they "force" the operating system to execute their virus code by modifying the necessary fields of the file system.

2.2.3.2. Boot Sector

Boot sector viruses infect the boot sector of a disk, affecting the computer system during the start-up process. The boot sector is a small area on a disk that is read by the computer when it is booted. This virus' operating principal is based on the algorithms of starting an operation system upon power on or reboot. After the necessary hardware tests for memory and disks are run, the system loader routine reads the first physical sector of a boot disk and passes the control to it [Kas99]. These viruses are difficult to deal with because they are executed during the start-up process, before the system is able to load its antivirus software [Lud98].

2.2.3.3. Macro viruses

Macro viruses are programs written in macro languages built into some application programs, for example, spreadsheets. To propagate, these viruses use the capabilities of macro languages to help transfer themselves from one infected document or spreadsheet to another. Macro viruses for Microsoft Word, Microsoft Excel and Microsoft Office are the most common. A macro virus is possible if the macro language built into a system has the following capabilities: a macro program must be tied to a particular file, macro programs must have the ability to be copied from one file to another, and a macro program must be able to receive control without user intervention. [Kas99]

They are the first viruses to infect data files, rather than executables. Data files, to which macros are attached, provide viruses with a more effective replication method than executable files. The effects of macro viruses are growing; data files are exchanged far more frequently than executable files, through e-mail and the Internet.

2.2.3.4. Network

Network viruses utilize networking protocols and the capabilities of local and global access networks to multiply. Their main operating principle is the capability to transfer viral code to a remote server or workstation. "Full-scale" network viruses also have the ability to run their code on remote computers and "trick" users to run the infected files. [Kas99]

2.2.3.5. Other Types

Other viruses are not as well understood as the types mentioned previously. They are trickier in their methods of avoiding virus detection, and present great problems to antivirus analysts and programmers. These types of viruses would not be recognized by the current scanning and analyzing techniques of MERCURY.

One of these more complicated viruses is the stealth virus. A stealth virus attempts to evade detection by concealing itself in infected files. To achieve this, the virus intercepts system calls that examine the contents or attributes of infected files. The results of these calls are altered to correspond to the file's original uninfected state. For example, a stealth virus might remove the virus code from an executable when it is read, instead of executed, so that an antivirus program will examine the original, uninfected host program. [Lud98]

Another tricky virus is the polymorphic virus. These viruses are programmed to change their internal code each time they replicate, making them more resistant to detection. Their main component is the polymorphic generator, which is responsible for creating varying encryption and decryption routines. These routines are used to hide the existence of the virus by constantly changing the byte signature of the virus [Kas99]. Since the most common method of virus detection is scanning for viral byte patterns, as in MERCURY, the detection program would likely be easily defeated. If you take two instances of the same polymorphic virus, there are *no bytes* in common between them. MERCURY, like most viral scanners would not be able to capture a unique byte pattern, in order to detect this ever-changing virus.

2.3. Antivirus Programs

Antivirus programs are the most effective means of fighting viruses, but *there are no antivirus programs that guarantee 100 % protection from viruses* [AAV97]. Such programs do not exist because for each antivirus algorithm, it is always possible to suggest a virus "counter" algorithm; however, the opposite is also true. For any virus algorithm, it is always possible to create an antivirus; the cycle never stops! Furthermore, the impossible existence of the absolute antivirus program has been mathematically proven by Fred Cohen, based on the theory of finite slot machines [AAV97].

As futile as it may seem, it is important to study viruses in order to uncover their weaknesses. Once these vulnerabilities are discovered, antivirus methods can be exploited, and virus detection *improved.* The following sections describe detection issues, *current* antivirus methods, their functions and their deficiencies.

When speaking of virus detection methods, it is necessary to understand some terms used in this discussion. A "false positive" is defined as an uninfected object, in the case of MERCURY, a self file, that triggers the antivirus program inadvertently. Conversely, a "false negative" is defined as an infected object that remained undetected by the detection system. "On-demand scanning" refers to virus scanning that starts on the user's request; if fully implemented, MERCURY would exhibit this characteristic. In this mode the antivirus program remains inactive until the user invokes it from command line, batch file or system scheduler. "On-the-fly scanning" methods, however, are constantly checking for viruses. In this method, the antivirus program is always active; it resides permanently in memory and checks objects, without the user's request. [AAV97]

2.3.1. Purpose

Computer viruses are detected by antivirus programs through the exact or "fuzzy" pattern recognition of a sequence of bytes called a signature, suspicious changes to files, or heuristic monitoring of "normal" viral activity. A powerful antivirus program would likely use a combination of these techniques. In addition to detecting the existence of a virus, some antivirus programs are responsible for repairing the damage caused by viruses [Kep94].

Typically, a "virus expert" obtains information about a particular virus by disassembling it and analyzing the assembly code to determine the virus' behavior and the method used to attach it to a host program. When the analyst has precise knowledge of the virus' attachment method, he can construct repair algorithms for a large class of similar viruses. This implies the virus has probably already infected files, possibly destroying data. Additionally, this reactive, labor intensive process requires manual intervention.

This research builds a computer immune model as part of a larger computer health model, and hypothesizes that constructive induction may offer an approach to detecting computer viruses that is both proactive and automated. Machine learning potentially provides the automation, while focusing on the self that belongs on the computer. Suspicion of the nonself may provide the insulation against the invaders. Since this research is *only* concerned with detection of viruses and not the repair of the damage, the following sections address the methods of detection currently used.

2.3.2. Requirements

The effectiveness and efficiency of an antivirus program is determined and can be measured by its reliability, quality of detection, and speed of completion. [AAV97]

The *reliability* of antivirus programs is the most important criterion, because even the best detector becomes useless if it cannot finish the scanning process. Reliability measures the ability of the program to finish the scanning process without unnecessary technical intervention from the user. If the program is unable to finish, parts of the disks and files will remain unchecked, possibly leaving a virus in the system undetected. Since MERCURY is not a fully functional virus detection system, this requirement could not be fully assessed.

Virus detection *quality* is the next requirement, since the main purpose of an antivirus program is to detect and remove viruses. Any antivirus program is useless if it is unable to catch viruses, or does it with low quality. The "perfect" detector would have a very high detection rate, accompanied by low false positive and false negative rates. If an antivirus program causes many false positives errors, then its level of usefulness drops significantly. This situation would cause the user to either delete uninfected files or analyze suspicious files, triggering frequent false alarms.

MERCURY was designed to favor detectors that classified a large number of files correctly. Additionally, the overall set of detectors was evaluated to ensure that the entire set of files was classified. The quality measures for each detector are supplemented by the quality measures for the set of detectors, which rate the percentage of files classified by the entire set of detectors. Due to the MERCURY's "infancy," assumptions about the quality of detection would be premature. The detailed results of the testing are presented in Chapter Six.

The next important criterion is working speed. If a full system check requires several hours to complete, it is unlikely that most users are going to run it frequently.

Currently, a disadvantage to MERCURY is its time to completion. However, several optimization methods are presented in Chapters Six and Seven that could attenuate this deficiency.

2.3.3. Types

The most popular and effective antivirus programs are scanners. They are followed by integrity monitors because of their effectiveness and popularity. Most of the time, though, both of these methods are combined into one versatile and more powerful antivirus program [AAV97]. The following sections describe these different methods.

2.3.3.1. Scanners

The principle operation of antivirus scanners is based on checks of files, sectors and system memory, and search for known and new, unknown viruses [AAV97]. A scanner recognizes viruses through an exact or fuzzy match of a relatively short sequence of bytes occurring in the virus called a "signature." Matching a small portion of the virus is more efficient in time and memory, and it enables the system to recognize variants. The signature is determined through a manual and tedious analysis of the viral code by virus experts. In addition, once the new signature is determined, the database of signatures is updated, requiring each user to manually update the local copy of the database on a regular basis. Between updates, though, users may be left exposed to spreading viruses. Researchers at the High Integrity Computing Laboratory [KeA94] investigated methods for automating this process. Their computer immune system generated byte patterns that detected nonself and stored them for the scanner. *MERCURY also uses this method of antivirus detection.* Scanning was chosen over the other detection methods due to its programmability, modifiability, simplicity, and functionality. This research recognizes that signature scanning is *only a part* of a robust virus checker; the scanning method was utilized as a tool for the proof-of-concept of constructive induction applied to the virus detection domain.

Scanners are divided into resident programs that work on-the-fly; and non-resident programs that check the system only on request. In most cases, resident scanners provide better system protection, because of their immediate reaction to the appearance of virus, whereas nonresident scanners can only detect viruses when executed [AAV97].

Scanners, including MERCURY, have the common advantage of versatility, and common disadvantages of large virus databases and relative slowness of virus search.

2.3.3.2. Integrity Monitors

Integrity monitors operate by calculating checksums for disk files and system sectors. These checksums are saved to a database along with data about file sizes and the dates of last modification. On subsequent runs integrity monitors compare database information with currently calculated values. If the database entry for a file differs from the file's current characteristics, the integrity monitor reports file modification or possible virus infection [AAV97]. While this method can determine if a file is changed, it provides no information on the legitimacy of the change.

Integrity monitors cannot catch a virus immediately after its infiltration. Rather, it detects after a period of time, when the virus has already spread throughout the computer. Additionally, this method cannot detect viruses in newly arrived files, like e-mail attachments, since these files do not have a baseline checksum.

2.3.3.3. Behavior Blockers

Antivirus behavior blockers are memory resident programs that intercept potential virus danger and warn the user about it. Virus danger may be detected during write calls to executable files, boot sector writes or during operations that may be conducive for viruses to spread. [AAV97]

Blockers are able to spot and block the virus at the earliest stage of infection, which is useful when a virus repeatedly launches surprise attacks. However, the challenges to this method are virus algorithms that override a blocker's protection and a possibility of a large number of false alarms. There are some improved versions of blockers with greater versatility, but they also have problems of compatibility with standard hardware configuration of computers, and are difficult to set up and configure. All these reasons make behavior blockers extremely unpopular compared to other methods of antivirus protection.

2.4. Machine Learning Methods

An important issue in artificial intelligence is machine learning, which enables machines to adapt, in order to improve their performance [ABKS94]. Machine learning research can be divided into five areas: neural networks, genetic algorithms, instance based learning, analytic learning and inductive learning [LaS95]. Neural networks attempt to model the structure of the brain by representing knowledge as a multilayer network of units that spreads activation from input nodes through internal units to output nodes [LaS95]. Genetic algorithms model the process of chromosome mutation through a series of rules generated by combining and/or mutating aspects of existing rules. Instance based learning uses a database of specific cases and experiences that are matched to general problems. Analytic learning uses logic to solve a problem by proving it from the supplied background knowledge. Collections of these proofs are then compiled into rules used to solve similar problems. Finally, *inductive learning* attempts to search for the combination of rules and attributes that best describe the problem that is represented as a series of examples, both positive and negative.

The following sections focus on inductive learning, specifically constructive induction, since this method of induction will be used as the learning mechanism for MERCURY. The application of this learning method, and the components and processes of constructive induction used for the design, development and implementation of MERCURY are detailed in Chapter Five.

2.4.1. Inductive Learning

Induction is a form of automated machine learning that derives knowledge from the observation of positive and negative examples of a concept. It attempts to characterize this concept by carefully selecting attributes, which are characteristics describing the examples, and, when necessary, constructing new, more useful, attributes. By characterizing the concept, this type of learning explains the given examples and is useful for predicting the class membership of subsequent unseen examples. The goal of the constructive induction process is either characteristic learning or discrimination. Characteristic learning is the ability to describe the examples. Discrimination is the ability to distinguish between positive and negative examples [Gun91]. MERCURY's learning component uses the discrimination form of induction to distinguish between self and nonself files on a computer system.

Inductive learning performs a series of manipulations that determines the *best* attributes needed to describe the current example set and distinguish between positive and negative examples. The end result of inductive learning is a description known as a *hypothesis,* a statement relating descriptive attributes of a concept to values those attributes assume in examples of the concept. A hypothesis indicates an approximation of the concept, subject to change, should future examples indicate it is incorrect. Collectively, these hypotheses form a *rule base* to describe the concept.

There are two forms of inductive learning: selective induction and constructive induction. For example, age, height, and weight can be the attributes to describe an instance of a person. To characterize the concept of "AFIT students," students would be used as positive examples and non-students as negative examples. Based on the attribute values of both the examples and counterexamples, a *hypothesis* could be made which *separates* students from non-students. If this can be accomplished by carefully selecting from the original attributes, this process is known as selective induction. If the original attributes are insufficient for this task, then new, more useful attributes are constructed; hence, the term constructive induction. In the previous example, if weight and height were insufficient attributes to characterize the concept of AFIT students, a new attribute could be constructed by combining one or more of them. For example, the new attribute could be BMI, Body Mass Index, which is calculated using weight and height. The final collection of hypotheses that best describes the concept forms the *rule base*. The following sections describe the components of an induction system and the two forms of inductive learning.

2.4.1.1. Components of Induction

An induction system is composed of several components: a representation of the problem, a set of examples that are described by the representation, rules that are used to construct new hypotheses, and background knowledge called bias. Selective and constructive induction differ by the process each uses to generate hypotheses. Selective induction is highly limited and the construction occurs in the build up of the final representation.

Representation is how the world is presented to the system. There are two key components of a representation: attributes and representation structure. Attributes are normally defined by inspection of the problem domain and information gained from experts. For example, the attributes used in this virus detection domain problem are bytes from a file. Determining the "best" attributes that allow the induction to work properly is the most important representation issue for inductive systems. This can be a complicated process, since the proper attributes may not be readily apparent to either the system developer or domain experts. There are varieties of different representation structures that can be used for maintaining the attributes inside of the system. Common methods include predicate calculus, decision trees, semantic networks and frames. [DiM81]

The importance of choosing the right attributes in the early stages of learning is a crucial concept in the implementation of MERCURY. Though not a *fully* developed constructive induction engine, MERCURY should benefit tremendously from prior knowledge of the virus domain, as well as the performance of features and operators used for the construction of new attributes.

An example set is used to initially train the system. Since induction relies upon examples to test if the system has generated an appropriate hypothesis, the types of examples used are important. The example set can normally be divided into two categories, each with two subcategories. The first division should divide the examples into a category used for training and a category used for testing. Training examples are used to perform the induction process. Testing examples are utilized to analyze how effective the process really was. Both training and testing example sets should include positive and negative examples. Negative examples should include examples that are both very different from and very close to the positive examples, in order to "fit" the best rule for defining the concept.

The number of examples needed is an important decision. The probability distribution describing the problem space is often difficult to determine for several reasons. The experimenter may have limited knowledge of the concept; the number of possible examples may be limited; or there may be an improper division of the example set, resulting in a statistically different distribution than the problem space.

Selecting the induction rules utilized by the system is also an important factor. These rules guide the selection of appropriate hypotheses, which form the best rule base. The varieties of inductive rules are subsequently discussed in the hypothesis generation section.

Bias is any factor that influences the hypotheses that are generated and evaluated. This factor is intrinsic to the inductive process, based upon the incorporation of the representation and the inductive algorithms utilized. Bias is information developed through common sense or derived by expert knowledge about the concept domain; it can decrease the time the system spends working with hypotheses that are later discarded. Bias is a domain specific issue and can rarely be generalized to all systems. [Pro92]

A bias can affect the learning time, space usage and accuracy of the system. Therefore, when choosing a bias the developer must make certain tradeoffs among the factors of the system. One tradeoff is the accuracy of learning versus the learning time. For example, as the number of attributes available to the system increases, so does the induction learning time. Another tradeoff is accuracy of learning versus the efficiency of space usage. A bias, such as a limiting the number of hypotheses, may inadvertently discarded the correct solution. [Pro92]

2.4.1.2. Selective Induction

Selective induction determines a combination of attributes that best describes the concept. Selective induction hypotheses are *limited* to applying the conjunction, disjunction, and negation operators to the given attributes. The conjunction operator joins two or more propositions through a logical *and*; both propositions must be true for the conjunction to be true. The disjunction operator joins two or more propositions through a logical *or*, one of the propositions must be true for the disjunction to be true. Finally, the negation operator states that a particular proposition can not be included in the concept. The selective induction process does not create new attributes, but instead combines existing ones. Selective induction generalizes conjuncts of attribute values or partitions instance space into contiguous regions. Conversely, constructive induction transforms the original representation by associating diverse conjuncts or dispersed regions that may appear unrelated [Ren90].

2.4.1.3. Constructive Induction

Constructive induction, like selective induction, determines a combination of attributes that best describes the concept. However, the constructive inductive process creates hypotheses by constructing *new* attributes. The process applies a variety of operators to the original attributes from the representation. These hypotheses are tested against the example set and incorporated into the rule base, if appropriate. The following sections describe the processes necessary to develop a constructive induction based learning mechanism.

2.4.1.3.1. The Constructive Induction Process

As previously stated, inductive learning depends on the ability of the machine to create hypotheses, indicating an approximation of the concept. Gunsch [Gun91] describes a framework for the inductive learning process that follows the path of hypotheses through the inductive system. The following sections describe the four processes of induction.

Hypothesis generation. The first step of the inductive learning process is the creation of new hypotheses based on current rules and attributes. Prior knowledge about the concept can help to explicitly define the hypotheses, or to guide the selection or retraction of operators to create new rules. An application of bias includes simplifying conjunctive rules, turning constants into variables, extending the range of an attribute, combining the attribute value intervals, and creating new rules that include exceptions. An exception can be found by looking for positive and negative examples that use similar attributes. For example if a positive example is described by $P(x)$ and a negative example by $P(x) \wedge Q(x)$ the exception is $P(x) \wedge \neg Q(x)$ [DiM81]. Other methods to induce rules involve applying mathematical operators (+,-,*,/) or logical operators (∧,∨,¬) to the attributes used in the representation.

Hypothesis ordering. Hypothesis ordering takes the generated hypotheses and performs a preliminary check to ensure that only the most promising of the hypotheses are evaluated. This step is a "heuristic, beam-sort approach and therefore does not guarantee perfect filtering" [Gun91]. The main purpose of this step is to limit the number of hypotheses to be evaluated, since hypothesis evaluation is an expensive process. This is a form of bias necessary to control the computational explosion generated by this learning method. Biases made during the development of MERCURY's learning component are discussed in Chapter Five.

Hypothesis evaluation. During hypothesis evaluation, each hypothesis is tested against the goal of the system to determine if it should be used. As described earlier, the goal of the constructive induction system is either characteristic learning or discrimination of positive and negative examples. If the hypothesis adequately meets the goal, then it is reserved for incorporation into the rule base. For example, hypotheses generated by MERCURY's learning component are evaluated based on their purity and power scores. Power measures the strength of the hypothesis by rating the percentage of classified examples over the entire set of examples, regardless of correctness. Purity calculates the percentage of examples classified correctly, over the total number of examples classified. If the hypothesis does not meet the goal, it is discarded or returned to the hypothesis generation step for additional inductive refinement. This step is computationally expensive since every hypothesis must be compared against all the members of the example set.

Hypothesis incorporation. In the final step, those hypotheses that have been determined to meet the system goals are incorporated into the rule base. The hypotheses are arranged within the rule base in the order they reduce entropy. Entropy is the amount of disorder remaining within the divisions of the examples after applying a hypothesis or group of hypotheses.

2.5. Machine Learning Applied to Detection and Recognition

Machine learning can help solve difficult, real-world problems which have well-defined tasks, implemented programs, and identified principles [Win92]. Intrusion detection, is one such problem; there are many applications of machine learning in computer security systems. The following sections describe the use of different machine learning methods to detect or recognize intruders, such as hackers or viruses, in computer systems.

2.5.1. Incremental Learning Applied to Computer Intrusion Detection

Research conducted at the George Mason University [MaM95] describes an incremental learning method, which modifies the current hypotheses set by combining past training examples with new ones. The proposed method is incremental because additional examples are incorporated into the training set over time. For this study, the process was applied to the domain of intrusion detection in computer systems. This domain was chosen computer systems needs to interact with users and the environment to adapt to changing conditions and behaviors. Researchers believe this method to be effective for applications involving intelligent agents in a changing environment, active vision, and dynamic knowledge bases.

The first phase of the methodology is traditional for the learning by examples method. It involved building an adequate training set, which provided the learning mechanism a sufficient concept of the computer environment. The second phase involved incrementally teaching the machine and customizing the concept to the specific environment of a particular user. The system received reinforcement or criticism from the user and/or environment. This approach uses feedback from the user or environment to determine training examples.

Concerns when applying this method are similar to concerns when a learning by example approach is taken. The choice of representative examples and the maintenance of these examples throughout the learning process are both important issues. Attributes considered important in the early stages of learning might not be important in later stages; more importantly, attributes considered unimportant in the early stages of learning might be important in later stages. For a rapidly changing domain, it is probably more beneficial to replace "old" attributes with "new and improved" attributes. For a more stable environment, it might be more practical to keep all attributes.

Compared to a batch method of providing all examples at once, the proposed method yielded significant gains in regard to learning time and memory requirements, but lost some predictive accuracy and gained concept complexity. Though the experiment was scaled down significantly, future work will involve applying this method to more complex problems.

MERCURY does not allow for incremental learning, but future iterations should. Virus detection, like intrusion detection, requires a degree of interaction with users and the environment to adapt to constantly changing conditions and behaviors. Future versions of MERCURY should allow for relearning when the definition of self changes, the system learns incorrectly, or the system receives virus prevention or detection information from other components in the Computer Health System.

2.5.2. Neural Networks Applied to Computer Virus Detection

Researchers at the IBM Thomas J. Watson Research Center [TKS96] developed a neural network for the detection of boot sector viruses, which was deployed commercially, as part of the IBM Antivirus software package. The team faced several challenges such as representation scheme, scarcity of training data, tradeoffs between false positive and false negative errors, and memory and computational complexity limitations.

They developed a representation scheme based on features of byte strings from infected and non-infected files, and using frequency analysis, narrowed the list of features to those whose probability of occurrence was higher. Their limited examples were divided into training and testing sets, with additional "man-made" inputs to combat generalization, or underlearning of the concept. For practical virus detection, more emphasis was placed on false positives, instead of false negatives; false positives are much more frequent. Memory and CPU constraints were reduced through changes in storage formats, but present future challenges to commercial use of larger mutilayer nets.

The neural net's performance performs as expected, having caught 75% of all new boot sector viruses since its release. Most of the viruses not caught eluded the system because their byte patterns were obscured in some way; improvements are constantly being made.

Similar to the system developed at IBM, MERCURY relies heavily on *a priori* knowledge. The results in Chapter Six show that statistical guidance can be used when developing the learning mechanism, reducing computational complexity and improving the method's efficiency.

2.5.3. Genetic Algorithms Applied to Recognition in the Immune System

Researchers [FJSP93] built a model to study the pattern recognition processes and learning that take place in the immune system, with a genetic algorithm as its central component. Antigens and antibodies, the major players in the human immune system were represented as binary strings. An antibody matches an antigen if their bit strings are complementary; however, the strings do not need to match exactly. Matching functions were defined, evaluated on their matching abilities, and awarded a fitness value.

In principle, the genetic algorithm should be able to find the matches with the highest fitness values by evolving a population of antibodies. Though there was much "noise" in the experiment, the algorithm was able to detect common patterns for antibodies. It also did not overlearn the problem by maintaining diversity within its population. This algorithm for maintaining diversity is related to the ordering and evaluating mechanism commonly employed in learning classifier systems.

Again, this research stresses the importance of domain knowledge to guide the learning process and improve the classification results.

2.6. Summary

This chapter presented a description of the areas needed to develop a constructive induction based computer virus detection system. The literature review described the function, structure, and types of computer viruses and explained the inadequacies of the current methods used for their detection. Another aspect of a virus detection system is the learning mechanism, also discussed this chapter. It described machine learning methods, focusing on the method of constructive induction. Current research areas applying machine learning to computer security problems were also addressed.

Chapter Three describes the analogies that can be drawn between MERCURY, the constructive induction based virus detector, and certain characteristics of the human immune system. The chapter also describes how this computer immune system fits into a larger model of a Computer Health System. Chapters Four and Five provide general and detailed descriptions, respectively, of a constructive induction based computer virus detector.

3. Computer Immune Models

3.1. Overview

Chapter Two described current methods of virus detection, concepts of constructive induction, and research trends utilizing machine learning for computer security and detection. This chapter explains some analogies between computer systems and human systems by comparing the mechanisms used for the local detection and global protection against virus invaders. The two computer models are the Computer Health System, based on public health system, and the computer immune system, based on the human immune system. Therefore, the first two sections provide discussions of these immunologically-based areas. In addition, research trends in the area of computer immunology are addressed. Finally, the computer models are presented. Their characteristics and components are defined individually, and their requirements are discussed. Special attention is afforded to MERCURY, the virus detection component of a computer immune system. Chapters Four and Five provide general and detailed descriptions of MERCURY, respectively.

3.2. An Overview of the Public Health System

Public health is the science and art of preventing disease, prolonging life and promoting health through an organized community effort responsible for: sanitizing the environment, controlling communicable infections, educating the individual in personal hygiene, organizing medical and nursing services for the early diagnosis and preventative treatment of disease, and developing a standard of living adequate for the maintenance of health. [Tur97]

The public health system is a social enterprise that utilizes current knowledge in order to have the maximum impact on the health of a population. It identifies problems that call for a preventative "team" approach to protect, promote, and improve health. Public health is a *collective effort* to identify solutions that prevent and avoid health problems.

Public health's prevention efforts in the form of social policies, community actions and personal decisions are responsible for 25 years of the 30-year improvement in life expectancy since 1900 [Tur97]. Economic benefits of the public health system can be measured in terms of prevention treatment over cure treatment cost savings.

3.2.1. Functions of the Public Health System

The public health system has three core functions: assessment, policy development and assurance, described in the following sections [Tur97].

3.2.1.1. Assessment

The public health system assesses the health needs of the community by establishing a systematic process that periodically provides information on the health status and health needs of the community. The system also investigates health hazards in the community by conducting investigations that identify the magnitude of health problems, duration, trends, location, and populations at risk. The third aspect of assessment involves analyzing the elements of identified health needs in order to find causes and contributing factors that place parts of the population at risk of adverse health outcomes.

3.2.1.2. Policy Development

This system function advocates public health, builds constituencies, identifies resources in the community and generates relationships with public and private agencies for the effective planning, implementation and management of public health activities. The system must set priorities among health needs based on the size and severity of the problems and the acceptability, economic feasibility and effectiveness of intervention. Another aspect of this function is the development of plans and policies addressing priority health needs and establishing goals and objectives that focus on local community needs.

3.2.1.3. Assurance

The third function involves managing resources and developing organizational structure through the acquisition, allocation, and control of human, physical, and fiscal resources. This includes maximizing the operational functions of the local public health system through the coordination of community agencies and efforts to avoid duplication of services. The system must also implement programs and other arrangements to ensure direct services for priority health needs identified in the community. Additionally, this function must evaluate programs, provide quality assurance, ensure programs are consistent with plans and policies, and provide feedback on inadequacies and changes. Informing and educating the public on community health, promoting an awareness about available public health services and promoting health education initiatives are an important aspect of this function because they contribute to individual and collective changes in health knowledge, attitudes, and practices.

3.2.2. The Public Health System Framework

To understand what public health represents and how the components relate to each other, a conceptual and understandable framework is necessary. This framework brings together the mission and functions of public health in relation to the inputs, processes, outputs, and outcomes of the system. The following table summarizes these components [Tur97].

Table 2 -- Public Health System Components

Inputs or Capacities	*Community leadership, human resources, fiscal ,and physical resources, information resources, and system organization necessary to carry out the core functions of public health*
Practices or Processes	*Organizational practices or processes that are necessary and sufficient to ensure that the core functions of public health are being carried out efficiently*
Outputs or Services	*Health services intended to prevent death, disease, and disability and to promote the quality of life*
Outcomes or Results	*Indicators of health status, risk education, and quality-of-life enhancement*

The mission of the public health system is to promote physical and mental health and prevent disease, injury and disability. The core functions of public health are assessment, policy development, and assurance. The inputs carry out the core functions through processes called public health practices. Inputs are subjected to these processes, resulting in outputs, in the form of activities labeled as programs or services. Outputs are intended to produce the desired results, characterized as health outcomes. Figure 3 shows the framework for public health; it attempts to bridge the gap between what public health *is* and *does* and *how* it does this. This framework tends to be very ambiguous and abstract, even to those in the medical field; however, most recognize the need for some type of formalization. The following sections examine several of the public health components in detail.

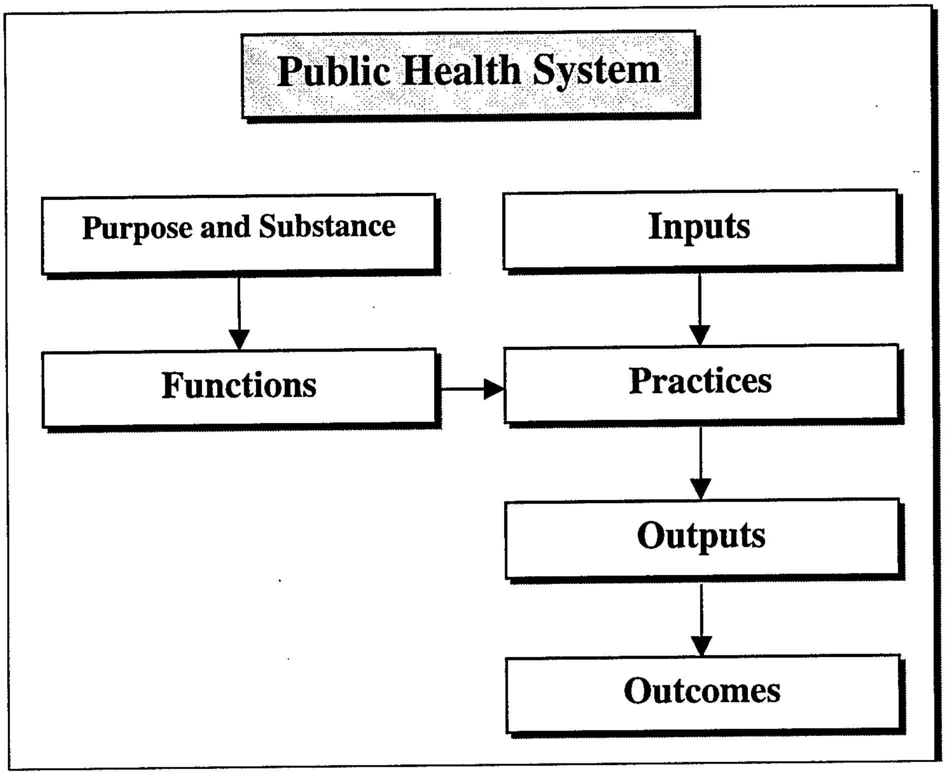

Figure 3 – The Public Health Framework

3.2.2.1. Infrastructure as an Input

Infrastructure can be described in terms of both static and dynamic attributes. In a static representation, the public health infrastructure is the basic building block and foundation for public health activities. In a more dynamic representation, the public health infrastructure is the *capacity* or capability of that foundation to carry out its main functions. The infrastructure serves as the *nerve center* of public health and represents the capacity necessary to carry out the core functions.

The infrastructure can be broken down into individual components. Components of the infrastructure are human resources, organizational resources, informational resources, and financial resources [Tur97]. The following table summarizes these components.

Table 3 -- Public Health Infrastructure Components

Human Resources	*Include the work force of public health and their knowledge, skills, and abilities*
Organizational Resources	*The relationships among the various system participants, public and private, and the mechanisms that manage the system practices, including their leadership components and collaborative strategies*
Informational Resources	*Include the various data, information, and communication systems*
Financial Resources	*The funding levels and sources for the work of public health*

The first of the components is the "workforce" of public health. The organizations facilitate the contributions of the workforce. Organizations are groups of individuals linked by common goals and objectives. This implies that each organization has a specific mission or purpose, resources appropriate to work toward that purpose, the ability to determine progress towards its goals and objectives, and defined process for making decisions.

The relationships among the agencies, organizations, institutions, and individuals are informal and collaborative rather than formalized and centrally directed. The workforce, the organizations, and their leadership rely heavily on information for identifying problems, determining interventions, and tracking progress toward common objectives. Together, these essential components of the infrastructure formulate the system's role in public health. [Tur97]

3.2.2.2. Essential Health Services as Practices

The following list summarizes the organizational practices that are necessary and sufficient to ensure that the core functions of public health are being carried out efficiently [Tur97].

1. Monitor health status to identify community health problems
2. Diagnose and investigate health problems and health hazards in the community
3. Inform, educate, and empower people about health issues
4. Mobilize community partnerships to identify and solve health problems
5. Develop policies and plans that support individual and community health efforts
6. Enforce laws and regulations that protect health and ensure safety
7. Link people with needed personal health services and ensure the provision of health care when otherwise unavailable
8. Assure a competent public health and personal health care workforce
9. Evaluate effectiveness, accessibility, and quality of personal and population-based health services
10. Research for new insights and innovative solutions to health problems

3.2.2.3. Health Care Delivery as a Practice

The supply of health care resources is another important component of the public health system. The growing numbers and types of health delivery systems reflect the recent changing environment. Increasing competition, combined with cost containment initiatives, has led to the generation of group medical practices, health maintenance organizations, preferred provider organizations, ambulatory surgery centers and emergency centers. Many of these delivery system have used managed-care strategies and methods that seek to control the utilization of services to reduce costs. [Tur97]

3.2.2.4. Prevention as a Practice

One of the most important features of public health is its reliance on prevention. Prevention characterizes actions that are taken to reduce the possibility of something happening, or minimize the damage if it does happen. Prevention is considered by many to be the purpose of public health.

Prevention intervention strategies are divided into three types: primary, secondary, and tertiary [Tur97]. Primary prevention involves the prevention of the actual disease or injury, by reducing the exposure or risk level factors. Secondary prevention attempts to identify and control disease processes in their early stages, before symptoms are apparent. Tertiary prevention seeks to prevent disability by restoring individuals to their optimal level of functioning following some kind of damage. Intervention at the primary, secondary, and tertiary levels is a dependant upon knowledge, resources, acceptability, effectiveness, and efficiency.

3.3. An Overview of the Human Immune System

Immunology is the study of the body's resistance to invasion by other organisms. The immune system uses several layers of defense to protect the body against invaders, known as pathogens. Initial barriers to infection are the skin and physiological barriers such as pH and temperature. If the pathogens are able to get past these barriers, they must be dealt with by another layer of the immune system. This section provides an engineer's perspective of this immune system layer by describing system components, types of immunity, immunological functions, processes used to perform these functions, and challenges to the immune system.

3.3.1. Immune System Components

One of the major components of the immune system is the lymphocyte. These white blood cells attack inflamed, infected cells. There are three major types of lymphocytes involved in the immune response. Two of these cell types are "born" as the same type of lymphoid cell and later differentiate in separate areas of the body. One line matures in the thymus and is referred to as a T cell, while the other line matures in the bone marrow and is called a B cell. These two types differ greatly in function, yet both have a similar purpose: *to recognize and react to specific pathogen targets in the body.* Lymphocytes are constantly circulating in the bloodstream and the lymphoid tissues, allowing a rapid sampling of all cells that might possess "receptors" for pathogens, or the ability to "bind to" pathogens.

Antigen-presenting cells (APC) are the third type of lymphocytes that participate in the immune response. Although these cells do not have receptors as the T and B cells do, they are still actively involved in the immune process. Their function is to "process" the pathogen from inside a cell, and "present" it on the surface, making the pathogen "visible" to the T and B cells [BSL96]. The following table summarizes the responsibilities of these three types of lymphocytes discussed in the following sections.

Table 4 -- Immune System Lymphocytes

B Cells	*Responsible for the production of antibodies, which enhance and activate various capabilities of the immune system*
T Cells	*Responsible for regulating antibody production and cellular immune reactions, and killing infectious cells*
APC Cells	*Responsible for processing and presenting antigens on the cell's surface for recognition by other immune cells*

3.3.1.1. B Cells

These cells develop in the bone marrow and produce antibodies, or protein molecules that enhance and activate various capabilities of the immune system. Antibodies mark infected cells for attack by other immune cells. This is the simplest immune response, yet very rapid and effective. Additionally, B cells mediate the immune response. Cells develop through the process of negative selection whereby they die unless they receive the “survival signal” from the environment. The “survival signal” is produced *only* when the antibodies do not react to the body. [WeC93]

Each B cell carries the antibodies on its surface that detect a unique antigen. After being stimulated by both antigens and T cells, they may return to the bone marrow to undertake their final maturation. Mature B cells do not secrete antibody, but instead differentiate into antibody-secreting plasma cells during antigen stimulation [WeC93].

B cells have three purposes: serve as the first line of defense against pathogens, specialize in neutralizing toxins, and secrete mucus to help create a barrier against infectious agents [Nos93].

3.3.1.2. T Cells

T cells develop in the thymus and are responsible for regulating antibody production and cellular immune reactions, and killing infectious cells. While in the thymus, these cells undergo a rigorous elimination process, akin to “boot camp.” Developing T cells are exposed to self: those that do not react to self may leave the thymus and take up residence throughout the body [WeC93]. Otherwise, they are eliminated.

There are two types of T cells: CD4 (helper or inflammatory) and CD8 (killer). The CD4 cells promote inflammation and signal other T cells to multiply. They also help B-cells by signaling an infection. CD8 cells are able to “punch holes” into the target cell and inject it with chemicals, killing the infected cell.

T cells differ from B cells in the *kind* of antigen they recognize and in the *way* they recognize an antigen. T cells are unable to recognize the entire antigen, but instead can recognize fragments of antigens, known as peptides. [Nos93]

3.3.1.3. Antigen Presenting Cells (APC)

The third type of lymphocyte is the antigen presenting cell, which processes and presents antigens to helper T cells. APCs include various types of lymphocytes, mainly B cells, macrophages, which "ingest" infected cells, and dendritic cells. These cells take in antigens and break them down, so fragments of the antigen can be brought to the cell's surface. These fragments are carried to the surface by MHC proteins.

Major Histocompatibility Complex (MHC) is a set of proteins present in all cells that bind to peptides produced within the cell and bring them to the cell surface, where they can be recognized by the immune system. An important feature of MHC is a groove in its structure, which enables the protein to bind to a wide range of antigenic peptides. When the invader replicates *inside* a cell, MHC carries the short peptide chains from those viral proteins to the cell surface. The patterns of these peptides are called epitopes, which allow for the detection of multiple pathogens by a single lymphocyte. During an immune response, the presence of these foreign peptides in the MHC groove tells the immune system that the cell is infected. [Jan93]

3.3.1.4. Complement

In addition to the three types of lymphocytes, the body utilizes other components to fight off infection, such as the complement system. Complement is a group of at least 11 proteins that circulate in the blood in an inactive, non-functional form. This system "complements" the activity of the antibodies in destroying bacteria, either by easing phagocytosis, or "eating of infected cells," or by puncturing the bacteria cell wall. It is an essential player in the adaptive immune system because it entails the production of molecules that influence cellular immune mechanisms.

3.3.2. Types of Immunity

"Immunity refers to all the mechanisms used by the body as protection against the environment agents that are foreign to the body" [BSL96]. Examples of agents, or pathogens are toxins, pollen, drugs, viruses, bacteria, and parasites. Pathogens can infect cells through two methods: the pathogens are *found within the membrane based organelles* through which they entered, or the pathogens gain access to the *fluid part of the cell* and the cell nucleus. Viruses are the most common intracellular pathogens. The body uses two types of immunity: innate and adaptive.

3.3.2.1. Innate

Innate immunity represents the part of the immune system present at birth, and helps the body "resist infection through normal body functions." It includes body surfaces, internal components, and other physiological barriers such as the skin, mucus membranes, pH and temperature. All of these elements affect pathogens directly or encourage other immune responses. This is a static system, unable to adapt to new invaders. One of the main components of innate immunity is the macrophage cell, responsible for the ingestion of foreign invaders.

The innate immune system is responsible for providing a barrier against infection and detection of *extracellular infections*. These infections occur when the pathogen is not yet bound to a specific cell in the body. The body defends against these infections through a multilevel defense that includes: phagocytosis of bacteria and other invaders by white blood cells and cells of the tissue macrophage system; destruction of organisms by the acid secretions of the stomach and by the digestive enzymes; resistance of the skin to invasion by organisms; and, presence in the blood of certain chemical compounds that attach to foreign organisms or toxins and destroy them [Guy81].

The innate immune system *generates* detectors such as macrophages and the complements. Since these detectors are not specific, there is no need for them to undergo a process of testing for reaction against self. The complement component of the innate immune system *detects* by an affinity to chemically react with bacteria. This reaction *coordinates* the complement system with the macrophages by signaling the macrophage to *destroy* the bacteria [RBM98].

3.3.2.2. Adaptive

Adaptive immunity enables the body to recognize and respond to previously unseen invaders. This is one of the most powerful capabilities of the immune system. *Immunity is learned upon contact with the offending pathogens.* Once an invader has been detected in the body, immune cells activate to learn the structure of and destroy the pathogen. An important feature of adaptive immunity is the body's ability to *remember previous invaders* allowing a faster response time upon subsequent encounters with the same pathogen.

The response begins with the B cells circulating throughout the bloodstream. The antibodies present on their surface have a high affinity to bind to specific antigens. When a B cell confronts its matching antigen, the antibodies on its surface bind to the antigen. Detection is founded upon recognizing epitopes. Following detection, the complement system is activated to destroy the antigen with the help of macrophages. Additionally, the B cell replicates with a large number of mutations. Through the process of natural selection, those B cells possessing the best antigen detecting capabilities are stored in immunological memory for future defense.

The adaptive immune system requires an elaborate *generation* process to produce detectors. The output of this process is a collection of cells that individually detect a few pathogens but collectively provide the capability to detect numerous pathogens. *Detection* of the antigen occurs through different methods dependent upon whether the pathogen is extracellular or intracellular.

As discussed previously, extracellular infections are normally handled by the innate immune system. One of the primary activities of the adaptive immune system is eliminating *intracellular infections.*

When the pathogen gains access to the inside of the cell, MHC binds to the pathogen and moves it to the cell surface, where it is detected by B cells or helper T cells. Once the immune response begins, the helper T cells secrete chemicals called cytokines. These chemicals activate additional B and T cells, amplifying this cell mediated immune response [Elg96, Pau93]. The binding between the pathogen fragments and the lymphocyte is the first of two *coordinating* signals necessary for an immune response. A second signal is used to activate killer T cells. Killer T cells destroy the infected cell by eliciting apoptosis, a process that forces the cell to kill itself. Killer T cells can also release chemicals called cytokines that limit viral replication within a cell while the cell attracts macrophages and other phagocytes to destroy the cell.

Table 5 -- Types of Immunity

Innate Immunity	*Represents the static part of the immune system present at birth and provides a barrier against infection and detection of extracellular infections*
Adaptive Immunity	*Enables the body to recognize and respond to previously unseen invaders and eliminates intracellular infections*

3.3.3. Functions

The immune system performs several functions in order to defend the body against invasion.

3.3.3.1. Detection

The immune system protects the body through the detection of nonself patterns. The immune system's detection capability is very powerful since it is highly distributed, detects previously unseen invaders, and does not require an exact match between detector and pathogen. Lymphocytes circulate throughout the body and bind to foreign invaders, initiating the immune process. The immune system utilizes a distributed system of millions of detectors to fight invaders. The detection problem is a hard problem since there are on the order of 10^6 self patterns to distinguish from 10^{16} nonself patterns [FHS97].

3.3.3.2. Adaptation

The immune system incorporates mechanisms that enable lymphocytes to *learn* the structures of specific foreign proteins; essentially, the immune system evolves and reproduces lymphocytes that have high affinities for specific pathogens. The immune system adapts through a process called *affinity maturation*, which is essentially a process of mutation and selection [Zin96].

When a B cell is activated by binding to a pathogen, it secretes *antibodies*, inactivating pathogens or identifying them to other innate system defenses for elimination. After this binding, the B cell hypermutates, creating additional receptors. The immune system is constantly adapting through slight variations of successful receptors in pursuit of the most effective immune response [Jan93]. All B cells compete for available pathogens, with the highest affinity B cells being the "fittest" and replicating the most.

3.3.3.3. Memory

The body remembers previously seen invaders, speeding up the response to subsequent encounters. The first time an invader is encountered, the body launches a primary response that learns the structure of the pathogen. The immune system stores this knowledge in memory cells. Subsequent invasions result in the activation of these memory cells, providing a very specific and rapid response.

Table 6 -- Immune System Functions

Detection	*The ability of the immune system to recognize nonself utilizing lymphocytes circulating throughout the body*
Adaptation	*The ability of the immune system to learn the structures of specific pathogens; the immune system evolves and reproduces lymphocytes that have high affinities for these pathogens*
Memory	*The ability of the immune system to efficiently and effectively remember previously seen invaders and speed up the response to subsequent encounters*

3.3.4. Human Immune Processes

Innate and adaptive immunity have common processes that are used as a framework for a model of the immune system: generation, detection, coordination, and destruction. During the *generation* process, the immune system creates all its antigenic detectors through random mutations and combinations of genetic material. Those detectors that are not self-reactive are released into the body. During the *detection* process, detectors move about the body attempting to determine the existence of nonself. The immune system provides a highly distributed detection system with local coordination. *Coordination* is required to ensure that the proper immune response is taken following detection of the pathogen. The immune system uses various signals and chemical attractions to coordinate the components of the immune system. The main function of the immune system is to destroy invaders. *Destruction* can occur through ingestion of the invader by immune cells, inducing the pathogen to kill itself, ingestion of infected cells, and other processes. Destruction also occurs through the creation of inhospitable environments for pathogens through pH, temperature and mucous.

Table 7 -- Immune System Processes

Generation	*The process of creating antigenic detectors through random mutations and combinations of genetic material*
Detection	*The process of detectors moving about the body attempting to determine the existence of nonself*
Coordination	*The process of implementing the proper immune response following the detection of a pathogen*
Destruction	*The process of eliminating the invader by immune cell and other processes*

3.3.5. Autoimmmunity

While the immune system usually protects us against foreign invasion, sometimes the recognition capabilities falter, causing the body to make antibodies and T cells that attack self. This process is called autoimmunity.

Autoimmunity may be caused by an abnormal immune response to normal self-antigens, a normal immune response to abnormal self, or an abnormal immune response to abnormal self. Two major mechanisms lead to autoimmunity: a change of self, leading to the formation of new antigens, and exposure to antigens that induce cross-reactive antibodies [BSL96]. Other conditions can also lead to autoimmunity, such as a decrease in killer T Cells or helper T Cells, a dysfunctional MHC presentation of peptides on a non-APC, problems with lymphocyte production, and genetic and hormonal factors [Ste93].

Autoimmune diseases are either organ-specific or systemic and are a consequence of a dysfunction in adaptive immune system, caused by self-reacting T cells or antibodies. Autoimmunity may also be induced by exposure to antigens bearing a close structural resemblance to normal tissue components, called antigen mimicry. In this case, damaged is caused when the immune system cross-reacts with the normal tissue that has been mimicked.

3.4. Computer Immune System Research

As the complexity of computer systems increases to a level comparable to biological systems, an analogy between computer systems and immune systems is possible. Computer scientists hope that in studying the human immune system, new solutions will emerge to computer viruses and other security problems. Beneficial properties of a computer immune system include: detection of a virus in the host, isolation of the virus and classification based on its characteristics, location of infected resources within the host, repair of any damaged host resources, and storage of information on previously encountered viruses [MVL98].

Current immunologically inspired research investigates different methods of detection. One method built a computer immune system to detect computer viruses across hosts connected to a network. Another research group used a computer immune system for network intrusion and virus detection, by monitoring system calls to detect intrusion. This same group also studied the generation of detectors modeled after the T cell generation of the human immune system. A third group defined a distributed architecture for a self-adaptive computer virus immune system, and described the role of evolutionary algorithms and performance of intelligent agents in this system. Finally, another approach investigated information survivability, utilizing the public health infrastructure as a model for a computer immune system.

The following sections expound upon these different approaches and their respective models. Since the last portion of this chapter combines advantageous properties from each of the research areas to form a different computer immune approach, these characteristics will be noted throughout the sections.

3.4.1. The Digital Petri Dish

Jeffrey Kephart and Steve White conducted research at IBM Thomas J. Watson Research Center [KSCW97], developing a computer immune system to detect computer viruses across hosts connected to a network. In their system, each networked PC analyzed potentially infected files, and sent suspected infected files to a central computer. The detection methods employed by the computers modeled the human immune system. Detectors were generated in large numbers, and those known to flag abnormal activity were replicated throughout the system, much like immune cells with receptors matching a given antigen are stimulated to reproduce themselves. This provided stronger selection for good recognizers and increased the chance of generating computer immune cells that are matched to a particular invader.

In their system, when one of the networked PCs receives a suspected "infected" sample, it sends the file to another computer that acts as a "digital petri dish." A software program on this computer tricks the virus into infecting a "decoy," bringing the viral code out of hiding [KSCW97]. If a virus is detected, a signature is extracted through bayesian methods, and sent to the infected host and other computers on the network. These bayesian methods analyze frequency of byte patterns occurring in infected and uninfected files. With a recognizer and a repair algorithm appropriate to the virus, the extracted viral information can be added to the corresponding databases. If the virus is ever encountered again, the immune system will recognize it immediately as a known virus. Figure 4 shows the petri dish concept.

A computer with this immune system could be thought of as “ill” during its first encounter with a virus. However, on subsequent encounters, detection and elimination of the virus would occur much more quickly, for the computer could be thought of as “immune” to the virus.

This computer immune system is desirable and feasible. The technology is being integrated with IBM AntiVirus, Symantec’s Norton AntiVirus, and Intel LANDesk Virus Protect [IBM98]. Most of the necessary components are already in use in one form or another, some already exist in IBM AntiVirus itself. Others are presently in use in the virus laboratory, for updating the databases employed by IBM AntiVirus to recognize viruses and repair infected files. Judging from the relatively low false-positive rate of the IBM AntiVirus signatures, the detector algorithm’s ability to select good signatures is better than can be achieved by typical human experts.

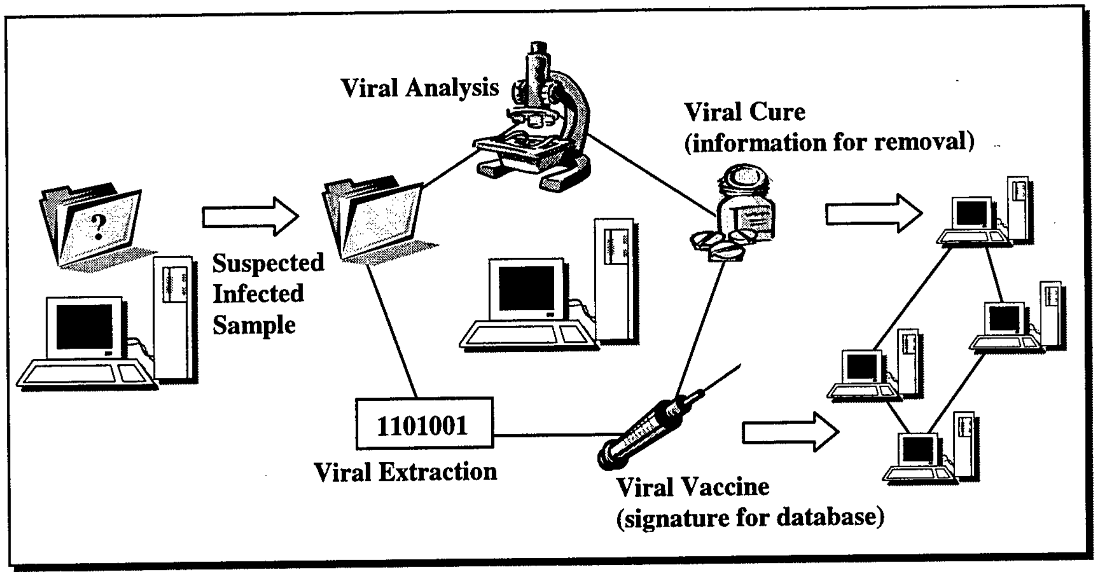

Figure 4 -- Pictorial Representation of the Digital Petri Dish

Table 8 -- Highlighted Features of the Digital Petri Dish

Advantageous Properties	
Network Connectivity	*Provides more efficient access to virus data and detection resources*
Centralized Viral Analysis	*Allows for easier coordination and less duplication of virus information and detection resources*
Information Sharing	*Provides efficient and effective means for disseminating virus information*
Disadvantageous Properties	
Network Connectivity	*Sensitive infected files are at risk when sent over the network; the system is dependant on the availability of network*
Centralized Viral Analysis	*Dependency is increased; single point of failure exists*

3.4.2. Improving Computer Security

Stephanie Forrest and researchers at the University of New Mexico, the Santa Fe Institute and Odyssey Research Associates conducted research that examined the similarities between living organisms and computers in order to improve computer security. Improvements in intrusion detection can be achieved by designing computer systems with some important characteristics taken from the human immune system: multi-layered protection; highly distributed detector, effector and memory systems; diversity of detection ability across individuals; inexact matching strategies; and sensitivity to most new foreign patterns [FHS97].

Their computer immune system has the following components: a stable definition of self; the ability to prevent or detect and subsequently eliminate dangerous foreign activities; memory of previous infections; a method for recognizing new infections; autonomy in managing responses; and a method of protecting the immune system itself from attack. [FHS97]

One of the first challenges the team confronted was the determination of self and nonself. They wanted their definition of self to be tolerant of legitimate changes, including those made to files, caused by adding new software or users, and routine activities of the system administrators. However, the system must be able to detect unauthorized changes and users, as well as viruses and inside attacks. These conflicting requirements were addressed in two supplemental areas of research: intrusion detection and distributed change detection.

3.4.2.1. Intrusion Detection System (IDS)

The IDS approach to security is based on the assumption that a system will not be secure, but that intrusions can be detected by monitoring and analyzing system behavior [HFS97]. In the network intrusion detection domain, their system is based upon self and nonself recognition through anomalous system calls. Each computer's definition of self is based upon a baseline analysis of system calls executed by privileged processes in a networked operating system.

Their strategy for the intrusion detection system was to first build up a database of normal behavior for each program of interest. Second, during a program's execution they scanned traces of system calls that might have contained abnormal behavior, and matched the trace against patterns stored in the database. If this trace did not occur in the normal database it was recorded as a mismatch, and used to distinguish between self and nonself. As in the body, the database of self is unique to each computer. Figure 5 presents the main concepts of this system.

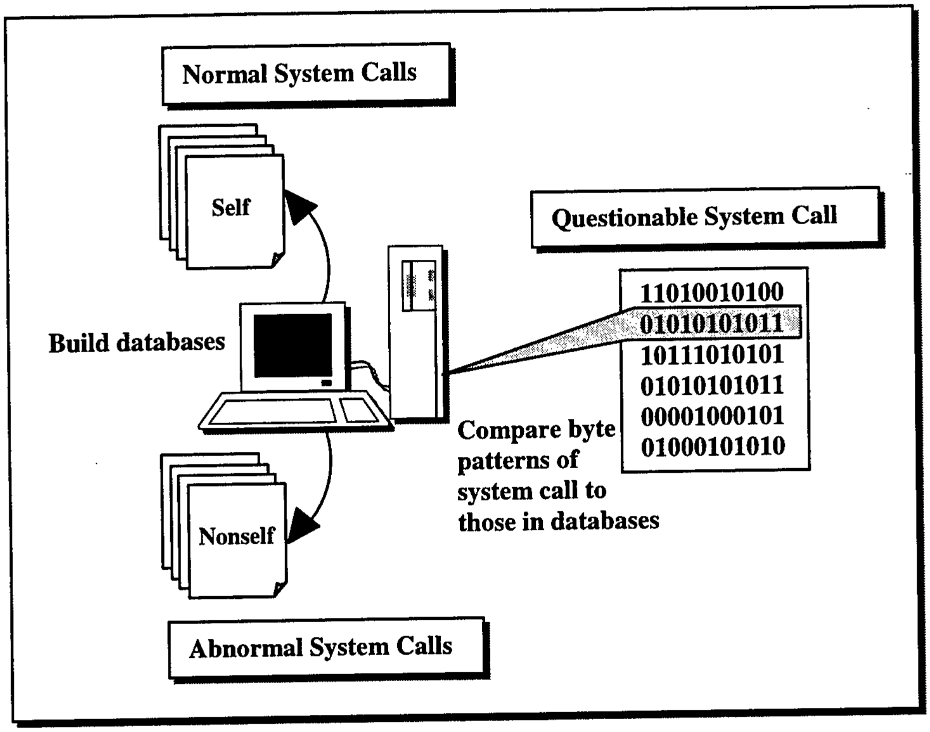

Figure 5 -- Pictorial Representation of the Intrusion Detection System

The research group constructed these databases, and performed abnormal traces for three Unix processes. Their results suggested that short sequences of system calls executed by running programs are a good discriminator between normal and abnormal operating characteristics. These calls provide a compact signature for normal behavior and the signature has a high probability of being perturbed during intrusions. [HFS97, FHS97]

According to the research team, the current system is far from having the capabilities of a natural immune system. Besides refining the notion of self on a computer, provisions need to be made to allow the concept of self to change over time. Much work needs to be done in the area of partial or approximated matching, for the team realizes they have no mechanism for self-adaptive learning, as in the case of affinity maturation or negative selection in the human immune system. [FHSL96]

Table 9 -- Highlighted Features of the Intrusion Detection System

Advantageous Properties	
System Specific	*The database of self is unique to each computer*
Decentralized Analysis	*Each system develops its own concept of self and adapts accordingly*
Multiple Applications	*Can be used to detect viruses, intruders, or malicious users*
Disadvantageous Property	
Lack of Learning	*There is no mechanism for self-adaptive learning*

3.4.2.2. Distributed Change Detection

This application of immunology borrows from mechanisms involving T cell generation and training. T cells have binding regions created through a random process, much like random strings could be generated. Since it is possible that these T cells will bind to self, they are tested before being leaving the thymus. This entire T cell censoring process can be thought of as defining a protected collection of self in terms of its complementary patterns of nonself. [FHS97] Figure 6 presents the concept of detector generation.

The research team designed a distributed change detection algorithm which: generates of a set of nonself detectors, uses detectors to monitor important data, and identifies the location of change when a detector activates [FHS97]. In their computer immune system, binding between detectors and foreign patterns is modeled as a match between two strings. Self is defined as a set of equal-length substrings, formed by segmenting the data, and each detector is defined as a string of equal length as the substrings.

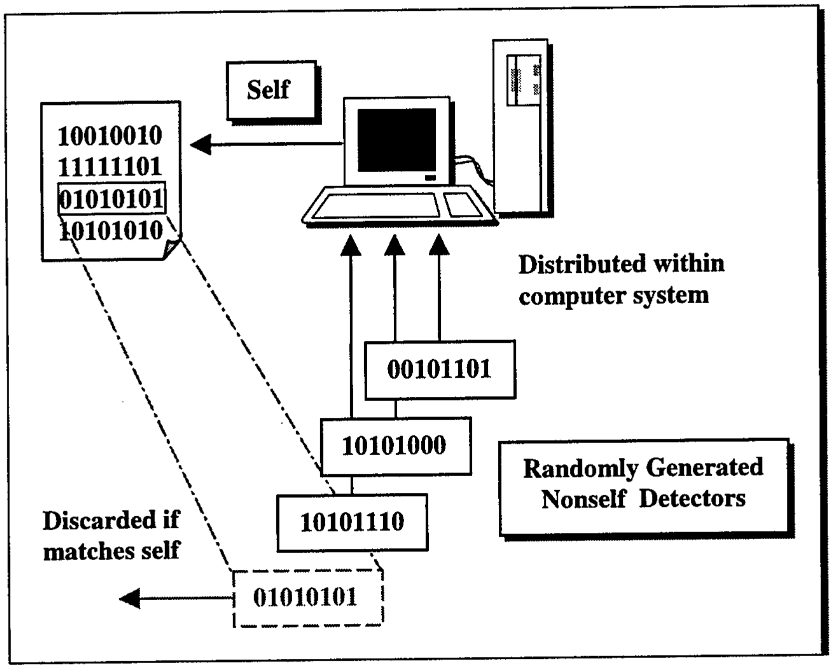
Self
10010010
11111101
01010101
10101010
Distributed within computer system
00101101
10101000
10101110
Randomly Generated Nonself Detectors
Discarded if matches self
01010101

Figure 6 -- Pictorial Representation of the Detector Generation

This system uses "negative detection," generating detectors for byte patterns not previously seen on the host [DFH96]. Detectors are continuously and randomly generated, compared against self, and discarded if they match. This approach is easily distributed because each detector covers a small part of nonself. A set of detectors could be split up over multiple sites. This would reduce the coverage at a given site, but would provide better system-wide coverage [FHS97]. To obtain similar results with self detectors would be much more expensive, because a complete set of positive detectors would be needed at every site, duplicating efforts and requiring much more communication between sites.

This algorithm is useful for dynamic or noisy data; it is effective for intrusion detection, as well as virus detection, the system's original intent. Based on the continuous detector generation and evaluation methods, this T cell based system could be especially advantageous when used as the adaptive component of a multi-layered computer immune system.

Table 10 -- Highlighted Features of the Distributed Change Detection

Advantageous Properties	
Distributed Detection	*The approach is easily distributed because each detector covers a small part of nonself; provides better system-wide coverage*
Adaptive	*Learning new concepts of self is easy due to nonself detector generation and evaluation*
Disadvantageous Property	
Incomplete Set of Detectors	*The randomly generated strings may not cover all possible combinations of nonself*

3.4.3. Distributed Architecture for a Self-Adaptive CVIS

Other researchers [LVM98] at the Air Force Institute of Technology are investigating a Computer Virus Immune System (CVIS). This system uses the human immune system as a model for identifying, attacking, and eradicating viruses from computers and networks. Based on an analysis of the requirements of such a system, this team proposed a distributed architecture that utilizes Intelligent Agents and Evolutionary Algorithms to self-adapt to a constantly changing virus population.

As pointed out by this research, there are several obstacles to implementing a CVIS. Though the human immune system provides a basic model for a computer counterpart, there are many “unmatched” parallels between the two systems. In the human immune system lymphocytes are distributed throughout the body and act as independent agents in search of invaders. This method has limitations in a computer immune system, such as: the number of available system processors, competition between tasks for processor time, and bottlenecks in accessing shared resources. [LVM98]

Their proposed CVIS requires distributed control, multi-layer security, diverse implementations, and self-adaptation in a dynamic software environment. All system activities should be autonomous and applicable to the various architectures of current computer systems. [LVM98]

Another challenge in the accomplishment of this type of system is the implementation of an artificial adaptation mechanism. Computer systems lack the evolutionary adaptation mechanism as seen in the human immune system. The development of several components within the CVIS is required, including virus detection, virus purging, and damage repair. This research led to the conclusion that computational complexity of such a task would be overwhelmingly difficult for one system to handle.

To overcome these challenges, this AFIT team proposed a multi-level distributed architecture with the responsibilities of managing the computational complexity associated with implementing this system. This was accomplished through the coordination of autonomous agents at three levels: local, network, and global. Each agent has a number of goals, a scope of competence, and the ability to collaborate and communicate with other agents, objects, and humans. Based on human immune system components, agents represent the following functions: detect, classify, repair, update, communicate, help, kill, and suppress [LVM98]. Figure 7 presents a pictorial view of this system.

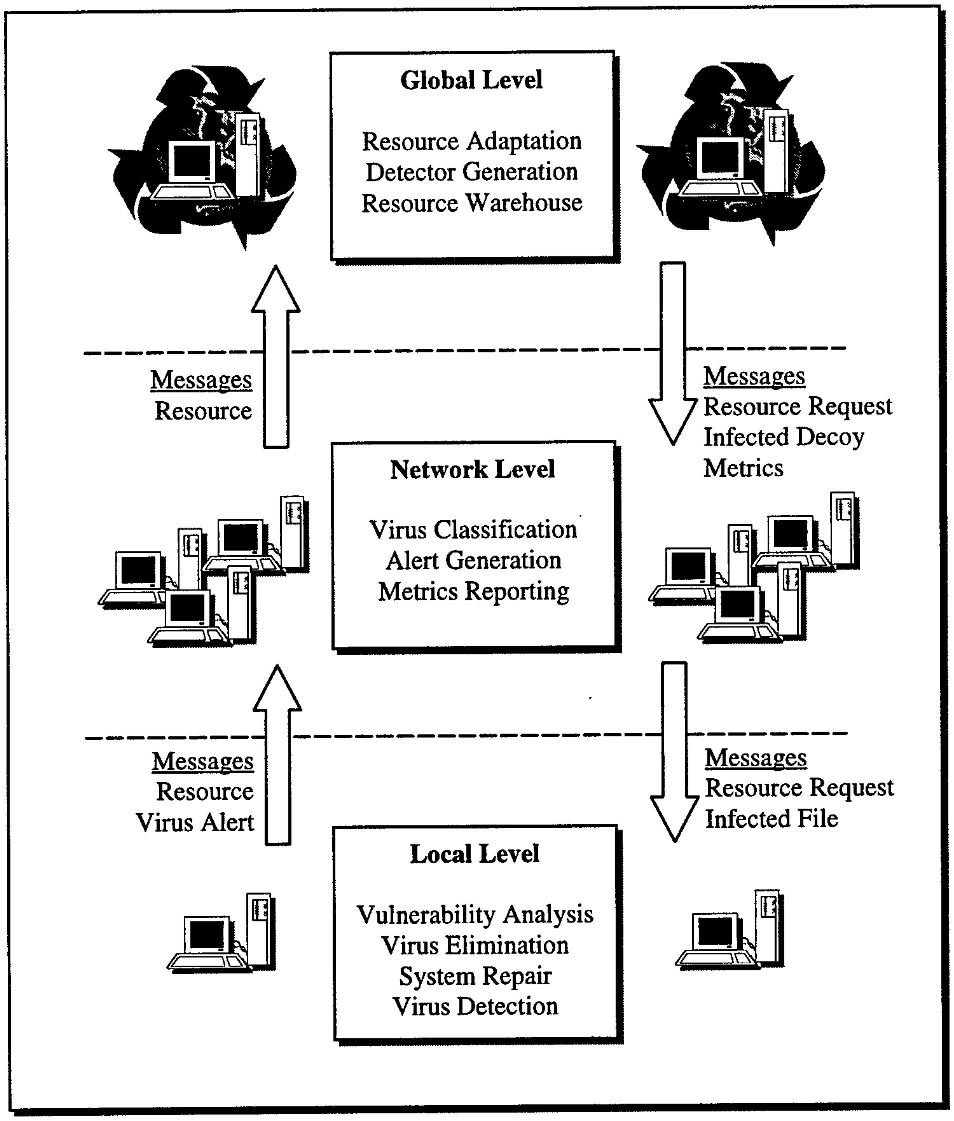

Figure 7 -- Pictorial Representation of the Self-Adaptive CVIS

Agents at the local level would be responsible for virus detection, virus elimination, system repair, and vulnerability analysis. Those agents at the network level are characterized by a high degree of interaction; they share many resources and are connected by a network. The network agents' purpose is to classify viruses, disseminate information and alert other agents of viral threats. At the global level, attention is given to the generation, adaptation and storage of virus information. Once developed, this information is disseminated to agents at the lower levels. Evolutionary Algorithms (EAs) are used in this CVIS at the global level for resource adaptation. They are used to improve the "decoy method" of virus detection, as originally presented by the researchers at IBM. The AFIT team's approach uses the networks linked to the CVIS as laboratories for virus evaluation. Using EAs enables a single, dedicated platform at the global level to mange a large decoy population.

This research combines previous human immune-based virus fighting efforts with the new twist of self-adaptation. The distributed nature of their system spreads out the computational burden of immune system tasks, and provides an efficient and effective CVIS.

Table 11 -- Highlighted Features of the CVIS

Advantageous Properties	
Network Connectivity	*A high degree of system interaction through the sharing of resources and connection by a network*
Compartmentalized Domain	*System responsibilities are divided among agents; agents function independently of each other*
Hierarchy of Responsibilities	*Agents are grouped into different levels and interact together to achieve the responsibilities of that level*
Information Sharing	*Provides efficient and effective means for disseminating virus information*
Adaptation	*Utilization of a learning mechanism to adapt to an ever changing virus population*
Disadvantageous Properties	
Network Connectivity	*The system is dependant on the availability of the network*
No Peer-to-Peer Communication	*Systems at the local level can not communicate with other systems at that level*

3.4.4. DARPA's Public Health Infrastructure

The Information Technology Office of DARPA presented research on information survivability, in response to the DoD's reliance on highly integrated and complex military information systems [Shr96]. Since these systems interoperate with the commercial computing infrastructure and rely on many of its components, penetration by unauthorized users can be conducted from virtually anywhere and by anyone. The goal of DARPA's research was to develop technology to ensure critical information systems continue to function adequately when attacked [Shr96]. These are large scale, complex distributed systems, many of which were developed without survivability as a prime concern. In order to improve current survivability approaches, this team looked to biological and social models, especially biological organisms, populations and society.

The benefits of individual organisms include barriers to infection and immune systems, detecting the presence of infection, and attacking and removing the invader. Other beneficial qualities of organisms include a redundancy and fault tolerance mechanism and homeostatic functions that maintain critical functions under stress. The team concluded these same ideas apply at the macro level of populations and societies. Collectively, the population is fault tolerant to the loss of individuals, and the species evolves based on which individuals survive the best in their environment. DARPA's research concentrated in three areas: public health infrastructure, adaptive architecture, and variability.

The *Public Health Infrastructure* element includes a set of security and survivability protocols with the ability to operate with available resources. The infrastructure supports problem solving between components of the system, such as: *decoys* deflecting attack, facilitating detection, and aiding with diagnosis; *canaries* warning of impending danger; and *honeypots* drawing attack to immaterial subsystems.

Just as a computer immune system would notice user intrusions into individual systems, corruption of data, or anomalous system behavior, this public health infrastructure must detect the symptoms of an attack effectively and efficiently. After symptoms are detected, the system must be able to disseminate this information rapidly and comprehensively. Once a part of the system is identified as infected, it must be "quarantined" so as not to infect other parts of the system. If the attacks increase, the system must be able to heighten its level of concern and allocate more resources for the diagnoses of the attack. As information about the attack is uncovered, vulnerabilities and recovery procedures must be relayed throughout the system.

Many issues accompany the public health model. There are many questions about the low-level immune system, especially building detectors, detecting attacks, learning anomalous behavior utilizing artificial intelligence, and keeping attack profiles current and correct. Issues involving information flow though the public health network include information control, method of flow, and methods for notification of attack.

The *Adaptive Architecture* element develops technologies that allocate resources to critical tasks, allowing the system to function while under attack. This system has a highly adaptive architecture to guarantee these functions continue with the loss of resources. This element also develops new models of semantic redundancy allowing corrupted data to be recovered by inference.

The *Variability* element develops technologies that allow differences between individual systems, in order to impede unknown threats. These technologies include: a variety of operating system implementations, randomized communication patterns, randomized memory layout, randomized allocations and variable operational patterns. Uniformity is dangerous; if system vulnerabilities were exploited, all machines would be immediately vulnerable.

These areas of public health infrastructure, adaptive architecture and variability together provide a high-level model of a Computer Health System; however, they fail to specify the lower-level details of an individual immune system.

Table 12 -- Highlighted Features of DARPA's Public Health Infrastructure

Advantageous Properties	
Secure Infrastructure	*Provides a set of security and survivability protocols with the ability to operate with available resources*
Component Communication	*All components can communicate with all other components through the system hierarchy (vertically) and peer-to-peer (horizontally)*
Information Sharing	*Provides efficient and effective means for disseminating virus information*
Disadvantageous Properties	
Unspecified Detection	*The ability of the individual detection system to recognize nonself is unspecified*
Unspecified Adaptation	*The ability of the individual detection system to learn the structures of specific viruses and adapt to the changing definition of self is unspecified*
Unspecified Memory	*The ability of the individual detection system to efficiently and effectively remember previously seen viruses is unspecified*

3.5. The Computer Models

The final portion of this chapter describes the two computer models developed as part of this research, which utilized some advantageous properties of the models discussed in the previous sections. The first model is a high level view of a Computer Health System, utilizing the public health system as a model. This Computer Health System is utilized for virus detection, though could be applied to other security domains. This health system includes the second immune model as a component. The second model is an individual computer immune system, with the virus detection component, MERCURY.

First, the general characteristics of model building are discussed, as well as the different types of models that can be used. Next, the Computer Health System is explained, from a high-level perspective of overall requirements and responsibilities. Finally, the computer immune system is presented, from a perspective of system properties. Throughout this explanation, parallels to the public health system and to the human immune system are drawn.

3.5.1. Model Building

The purpose of building a model is to explain observations made at particular levels for a given experiment and to possibly serve as a device for predicting what may occur at a specified spatial or temporal point of interest [Cas97]. A good model is not required to capture all aspects of the system it represents; rather, it should capture the system's *essence* [MVL98]. The model should be simple, clear, tractable, and relatively bias-free [Cas97].

Models can be either formal or informal, depending on their description or purpose. An informal model describes or represents system observations, components, or interactions imprecisely [MVL98]. It can help explain observations, but can't make accurate predictions of future ones. Formal models, however, are more likely to have a higher predictive power because they are mathematically based and represent the "true" system with greater accuracy. A formal mathematical system is a collection of abstract symbols with a set of rules expressing how strings of these symbols can be combined to form new strings [MVL98].

There are also different categories of models based on their purposes: predictive, explanatory, and prescriptive. Predictive models enable developers to predict future system behavior based on the properties of the system's components and their current behaviors. Conversely, the purpose of an explanatory model is to not predict future behavior of a system, but rather to provide a framework in which past observations can be understood as an overall process. Both predictive and explanatory models are passive, whereas the prescriptive model is active. The prescriptive model offers a top-level picture of the real world, enabling the developer to vary parameters for analysis. The following tables summarize the different types of models.

Table 13 -- Summary of Model Types

Formal	*Mathematical basis for representing the "true" system*
Informal	*Describes system observations, components, or interactions imprecisely*

Predictive	*A prediction of future system behavior based on the system's components and their current behaviors*
Explanatory	*A framework in which past observations can be understood as an overall process*
Prescriptive	*A top-level picture of the real world, with the ability to vary parameters*

3.5.2. Modeling a Computer Health System (CHS)

The CHS is an informal, explanatory model based on *some* essential qualities of the public health system. Due to the model's informality, not every aspect of the model will be explicitly stated. Though many parallels can be drawn between the systems, the CHS framework will not present the complete picture or solution. Rather, it is an approach to the overall process of detection, prevention, and cure of viruses on computer systems. Figure 8 presents the pictorial view of this system, its components, and their responsibilities and interfaces.

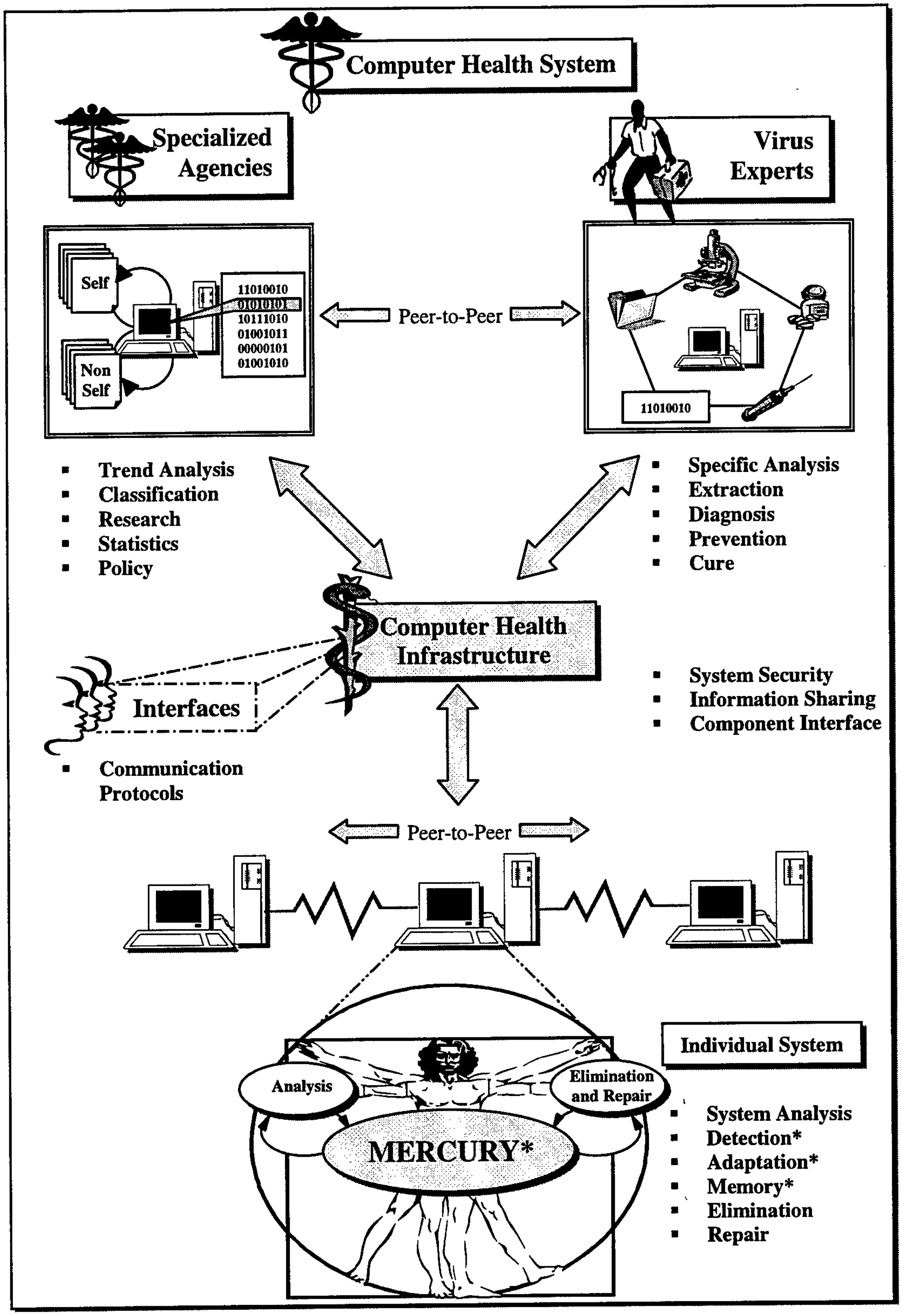

Figure 8 -- Pictorial Representation of the Computer Health System

3.5.2.1. Computer Health System Objectives

DARPA's research suggests the public health system is a highly robust and survivable infrastructure developed to detect, diagnose, isolate, cure, and prevent infections. The CHS presented here has the same overall objectives for the protection of computer systems against viruses.

Table 14 -- Comparison of Health System Objectives

Objectives of Public Health System	*Objectives of Computer Health System*
Detect Human Diseases	*Detect Computer Viruses*
Diagnose Human Diseases	*Diagnose Computer Viruses*
Isolate Human Diseases	*Isolate Computer Viruses*
Cure Human Diseases	*Cure Computer Viruses*
Prevent Human Diseases	*Prevent Computer Viruses*

3.5.2.2. Computer Health System Requirements

Like the public health system, the CHS is a social enterprise that utilizes current knowledge in ways that have the maximum impact on the way a computer system protects against virus invaders. This system identifies computer virus problems through a preventative "team" approach in order to protect, promote, and improve the "health" of computer systems, with an *emphasis on preventative strategies*. The primary goal of the CHS is to provide a framework for the globally scoped protection of computer systems against virus invaders. Some of the advantageous properties of the previously discussed immune models are included as requirements for the CHS. These requirements also improve upon the previously discussed models' shortcomings.

The CHS functions optimally, yet not solely with *network connectivity.* This connection provides the most efficient access to virus data and detection resources. Due to the individual immune systems on each computer, and many paths of connection throughout the network, this global system does not possess a single point of failure. Therefore, components can function autonomously with limited, or nonexistent network capabilities. Using a network connection precipitates the need for a *secure infrastructure.* This includes a set of security and survivability protocols that operate with available resources. These protocols ensure the protection of information as it traverses through the system's networks.

The system must provide *component communication.* All components must be able to communicate with all other components of the system. This communication includes hierarchical or "vertical" interchanges and peer-to-peer or "horizontal" connections. Component communication provides the vehicle for *information sharing.* This provides the essential means for efficiently and effectively disseminating computer virus information.

The system is partly based on *centralized viral analysis.* Specialized computer health agencies and virus experts are responsible for much of the computer health objectives. Their main tasks are explained in detail in subsequent sections. This allows easier coordination and less duplication of virus information and detection resources. Additionally, the system is partly based on *decentralized viral analysis.* Each individual computer immune system develops its own concept of self and adapts to viruses to the best of its ability, with the help of global prevention information and self-adaptability using MERCURY.

In addition to these requirements, the system also possesses intervention strategies extracted from the public health system. These intervention strategies can be divided into three types: primary, secondary, and tertiary.

In the computer sense, *primary prevention* involves the prevention of the actual virus, by reducing the exposure or risk level factors. This type of prevention includes regular disk and system scanning through an antivirus system, and responding to updated prevention measures, such as virus updates. *Secondary prevention* takes place if the virus had infected the system. This prevention would detect and control the virus destruction in its early stages, perhaps before the virus executed or caused large amounts of damage to the system. *Tertiary prevention* would involve repairing the system by restoring it to the optimal level of functioning following an infection. System intervention at the primary, secondary, and tertiary levels is a dependant upon the knowledge base of the computer immune system, the knowledge base of the virus experts, and the effectiveness and efficiency of the virus information transfer between components in the system.

Primary prevention reduces the number of virus infections, whereas secondary and tertiary prevention reduce virus predominance by decreasing the "life" of a virus and minimize its effects. As with public health, *approaches emphasizing primary prevention have greater potential benefit than approaches emphasizing other levels of prevention.*

3.5.2.3. Computer Health System Services

Many of the public health services translate easily into the computer health domain. The following chart states a few of the services this global system could provide.

1. Monitor virus status within the network communities
2. Diagnose and investigate viruses found on computer systems
3. Inform, educate, and empower users about virus issues and PREVENTION!
4. Mobilize research groups to identify and solve virus problems
5. Develop policies and plans that support virus detection and prevention efforts
6. Enforce laws and regulations that protect computer systems against malicious attacks
7. Link users and administrators with specialized agencies and virus experts
8. Assure a competent computer health workforce
9. Evaluate effectiveness, accessibility, and quality of the infrastructure and system
10. Research innovative solutions to virus problems

3.5.2.4. Computer Health System Functions

As in the public health system, the Computer Health performs three core functions: assessment, policy development and assurance. The Computer Health System *assesses* the needs of the computer systems and networks by establishing systematic processes that periodically provide information on the status of viruses through the cyber community. The CHS also investigates virus hazards in the community by conducting research that identifies the magnitude of viral damage, symptoms, and avoidance. The CHS must advocate collaborative research efforts, identify resources in the cyber community, and generate relationships with public and private agencies for the study of viruses. Another function of the CHS is the development of *plans* and *policies* addressing virus attacks and establishment of goals and objectives that focus on preventive measures. The CHS is responsible for informing and educating the users and administrators on computer health, promoting awareness about available antivirus applications and preventative techniques in order to increase knowledge, attitudes, and practices about computer viruses.

3.5.2.5. Computer Health System Components

Responsibilities are allocated among the four main components of the Computer Health System: specialized agencies, virus experts, infrastructure, and individual systems.

3.5.2.5.1. Specialized Agencies

Specialized agencies are organizations or research groups that facilitate contributions to the virus detection field. These agencies are linked by common goals and objectives with the specific mission of some aspect of virus detection or prevention. The relationships among the agencies are informal and collaborative rather than formalized and centrally directed. The main responsibility of these agencies is to conduct research and trend analyses of viruses, and develop useful statistics and metrics. Also, they provide a general classification for viruses, and formalize the methods of virus detection, extraction, and repair. Lastly, they provide policy guidelines and standards to establish goals and assign responsibilities within the Computer Health System.

The Center for Virus Control (CVC) is a proposed computer health agency based on the Center for Disease Control and Prevention. The main goals of the CDC are surveillance, applied research, and prevention and control. [OUSD96]. The CVC would be responsible for detecting, investigating and monitoring virus threats, and determining the factors influencing their occurrence. The agency would be responsible for integrating industry, academia and government research, improving virus security practices. The CVC would enhance private and government communication about emerging viral threats, and ensure prompt implementation of prevention and control strategies. Information dissemination would be a top priority, as would establishing implementation standards and guidelines.

The Virus Prevention Agency (VPA) is a proposed computer health agency based on the Federal Emergency Management Agency. The mission of this public health agency is to provide leadership and support to reduce the loss of life and property and protect our nation's institutions from all types of hazards through a risk-based, emergency management program of mitigation, preparedness, response and recovery [OUSD96]. Applying this to the computer health domain, the agency is responsible for helping network and computer systems defend against viruses, making virus repair assistance available to all users, and teaching users and administrators how to protect against viruses.

3.5.2.5.2. Virus Experts

Within the public health system, many physicians practice in groups where they can share expenses, medical equipment, and responsibility for emergencies. In the public health system, group practices, hospitals and other arrangements are called Health Service Organizations (HSOs) [Ayr96]. This same technique of pooling resources, personnel and knowledge can be applied to the virus expert component of the Computer

Health System. In this system the main responsibilities of the virus experts are: implementing and teaching preventative techniques, analyzing new viral types, extracting and diagnosing new viruses, and "curing" the system once it has been infected. These services provided by the virus experts are globally based and are targeted toward network systems rather then individual systems. Virus experts may be augmented with automated systems such as IBM's digital petri dish or MERCURY.

3.5.2.5.3. Infrastructure

Analogous to the public health system infrastructure, the infrastructure of the CHS is the backbone that protects and carries all the information flowing throughout the system. The supply of health care resources is an essential concern to the public health system; whereas, the supply of information is an important aspect of the Computer Health System. The CHS infrastructure provides the following functions: system security, information sharing and component interfacing.

The infrastructure must provide *system security* through three fundamental objectives. *Confidentiality* requires that the data in a computer system, as well as the data transmitted between computer systems, be revealed only to authorized individuals. *Integrity* stipulates that the data in a computer system, as well as the data transmitted between computer systems, be free from unauthorized modification or deletion. *Availability* requires that the authorized users of the computer systems and communications media not be denied access when access is desired. [WPF96]

The infrastructure also provides the CHS requirement of *information sharing.* Information is shared rapidly and comprehensively between agencies and experts, and among the population of computer systems. Virus information flows upward when a system detects or is infected with a virus and flows downward when a virus cure is discovered or preventative measures are formalized.

The *component interfaces* throughout the system require interfaces to manage the information control, method of flow, and methods for notification of attack. These concerns are analogous to the different types of health delivery systems available in the public health system. Group medical practices, health maintenance organizations and preferred provider organizations are all methods for individuals to "connect" to the public health system, just as *communication protocols between and within subsystems* enable the two-way connections between all subsystems and peer-to-peer systems.

3.5.2.5.4. Computer Systems

The final component of the Computer Health System is the individual computer system, equipped with a computer immune system. This immune system is similar to the human immune system, in that its purpose is to protect the computer system from invasion by viruses. The main functions of the individual computer immune system are system analysis, virus detection, self-adaptation, memory, and virus elimination/system repair. These functions are described in Section 3.5.1.

3.5.3. Modeling a Computer Immune System

Computer scientists hope that in studying the human immune system, new solutions to combating computer viruses will emerge. There are many properties of the immune system of interest to a computer scientist. Human immune systems are unique, meaning *individual* immunity is derived and adapted; this is a desirable and applicable property of a computer system, as well. The immune system uses an efficient decentralized and distributed detection process, which is also of interest in the virus detection domain. The human system is also very flexible and does not require the *absolute* detection of every invader; instead, *partial* detection allows for quicker recognition of multiple invaders. This is applicable to partial detection of byte patterns in an infected file on a computer system. Another very important feature of the immune system is its ability to detect and react to invaders, or nonself, while not reacting to what belongs in the body, or self. This property applies to invaders that have been previously seen, as well as those previously unseen. The human system can learn and remember the structures of these previously unseen invaders, so that the body's future responses to the same invader can be faster. [FHS97]

An individual computer immune system is an informal, explanatory model that captures the essence of the human immune system. Due to the model's informality, not every aspect of the model will be explicitly stated. Though many parallels can be drawn between systems, the computer immune system model will not present the complete picture or solution. While the CHS provides the framework for *global protection* against viruses, the computer immune system provides for *local detection* of these invaders.

3.5.3.1. Functions

As introduced in Section 3.5.2.4, there are several desirable functions for a computer immune system: system analysis, virus detection, adaptation, memory, virus elimination, and system repair. MERCURY, a prototype of a virus detection component of a computer immune system, is designed to incorporate three of these functions, detection, adaptation and memory. The following subsections describe each function, and, if appropriate, their design in MERCURY.

3.5.3.1.1. System Analysis

The system analysis function provides local prevention from computer viruses. Analysis is individualized through prevention strategies such as policy enforcement, disk checking and system scanning procedures, and analysis for viral system calls. Additionally, this function provides an analysis of normal system activity, in a manner similar to user profiling utilized in intrusion detection systems [WFP96]. Policies and procedures are obtained from the specialized agencies and specific preventive measures are obtained from the virus experts of the Computer Health System. MERCURY does not incorporate system analysis, but can be integrated with existing systems that provide this functionality.

3.5.3.1.2. Detection

Just as antigens can infect cells in the body through two methods, intracellular and extracellular, a virus can infect a system in two basic ways. The virus is *found within the system, such as memory, or boot sector* or the virus gained access to the *inside of a file.* Similar to the human system, a computer system detects a virus through different methods, dependent upon whether the virus is "extracellular" or "intracellular."

Extracellular Infections on a Computer. Similar to the multilevel defense against infections in the body, behavior blockers raise a warning when suspicious activity occurs on the computer. These programs have a sense of typical virus behavior, such as access to certain system resources and files, just as certain chemical compounds in the blood know to attach to foreign organisms or toxins and destroy them. Behavior blockers use a reactive method, since a virus is detected after the computer is infected.

Intracellular Infections on a Computer. Similar to the lymphocytes circulating in the bloodstream looking for pathogens found in the body, a virus scanning program would "circulate" through the computer system and compare system and data byte patterns to those previously seen patterns maintained in a virus database. In the body, the lymphocytes can not see the pathogen "inside" the cell without the assistance of MHC, which brings the invader to the cell's surface. In a computer, an infected file does not present the infection to virus scanner; instead, the scanner must look inside the file in order to determine if a file is infected.

The scanning component of MERCURY, though not fully implemented, would be responsible for extracting and detecting byte patterns in files. MERCURY's scanner recognizes viruses through an exact match of a relatively short sequence of bytes occurring in the virus, or by a rule generated by the learning component of MERCURY which specifies a set of bytes, combined using certain operators. Matching a small portion of the virus is more efficient in time and memory, and it enables the system to recognize variants.

As a component of MERCURY, a virus scanner was developed to evaluate the byte patterns inside files. This scanner functions just as the "team" of lymphocytes would in the human body. As the scanner reads the byte patterns in each file, they are compared to nonself and self byte patterns stored in the knowledge base. In the body, B cells circulate through the body, looking to bind with nonself antigens. If nonself is encountered, the cell is flagged as infected, and the "killer" cells are activated. In the computer system, if the pattern matches what is known to be a virus, an immune process is also initiated. The file is "flagged" and the elimination and repair function of the computer immune system is activated.

3.5.3.1.3. Adaptation

Detection involves the recognition of known viruses, whereas adaptation deals with previously unknown viruses. In the human immune system, *innate immunity* represents the part of the immune system present at birth. In a computer system, innate immunity can be thought of as what is initially learned, supervised or unsupervised, as acceptable system byte patterns and acceptable system activity. These allowable patterns and activities could be defined through initial heuristics and analysis. Similar to the work done by Forrest, databases of these allowable byte patterns and activities could be constructed.

Adaptive immunity for a computer system, like the human immune system, would enable the system to recognize and respond to previously unseen viruses. Once an previously unseen byte pattern is encountered in the computer, the system detectors determine whether it is a virus or new self, based on the predictive capabilities of current definitions of self and nonself. If the system detectors are unable to incorporate the byte pattern into these definitions, the system must seek advice from a virus expert. After the byte pattern has been identified by the expert, the system incorporates the byte pattern into the knowledge base so that it can be recognized quickly on subsequent infections.

As a component of MERCURY, the constructive induction engine, HEC was developed to perform the adaptive function of learning new byte patterns. On a fully operational system, once a byte pattern is labeled as self or nonself, HEC would incorporates it into the system's concepts of self and nonself, and update the knowledge base. As a computer changes, its immune system must adapt. Constructive induction provides a mechanism to incorporate these modifications and classify them as self or nonself.

3.5.3.1.4. Memory

A computer system must have the ability to *remember previous viruses,* allowing a faster response time upon subsequent encounters with the same virus. As a component of an operational version of MERCURY, a knowledge base could be built that contains the byte patterns that distinguish between self and nonself. This knowledge base would be accessed by the scanner and updated by HEC. MERCURY's current implementation, however, does not utilize memory external to the constructive induction engine.

3.5.3.1.5. Virus Elimination and System Repair

Once nonself byte patterns have been recognized in a file, the computer immune system must respond by enabling the elimination and repair function. This can be accomplished by attempting to reconstruct the file from a checksum database repository, or by attempting to identify and remove the exact viral code [KSSW97]. While the purpose of this computer immune function is analogous to the human immune system, the implementation differs. In the human body, cells are expendable, and can be killed without affecting the overall function of the system. However, in the computer system, files can not be "killed," or deleted with the same degree of indifference. MERCURY does not incorporate elimination and repair, but can be integrated with existing systems that provide this functionality.

3.5.3.2. Autoimmunity

While the purpose of this computer immune system is to protect against foreign invasion, the recognition capabilities of MERCURY could falter, causing the system to classify self as nonself. This process would be similar to autoimmunity in the human immune system.

For example, a common error occurs when the antivirus program reports "false positives" on legitimate programs. This results from the current manual techniques used to extract virus byte patterns from an infected file. These current antivirus techniques can be expected to fail within the next few years with the rapid, accelerating influx of new computer viruses.

Another problem with detecting viruses in a computer system is that the notion of self in computers is questionable. Self is not solely the pre-loaded software on a computer when purchased. Users are continually updating and adding new software, so it would be unacceptable if the computer immune system were to reject all such modifications and additions on the basis that they were different from what was on the system already. The human immune system can usually get away with "presuming the guilt" of anything unfamiliar, whereas the computer immune system must presume that new software is innocent until it can prove that it is guilty of containing a virus. [Kep94]

Autoimmunity could occur in two ways within MERCURY. The scanner could detect a previously unseen self byte pattern as nonself. Additionally, if the self and nonself files are similar, HEC could learn incorrectly. This could be treated proactively or reactively. Proactive treatment would require the user to interface with MERCURY, through the addition of new examples. The reactive treatment involves adding the new system files and waiting for the byte patterns in this file to be detected. After the new byte patterns are detected, they could be sent to HEC to relearn the self concept. Although an important function of a computer immune system, MERCURY does not allow for user intervention, but could be modified if needed.

3.5.3.3. Computer Immune System Interfaces

A computer immune system should contain the five functions discussed in the preceding sections. MERCURY was designed to incorporate three of these functions, detection, adaptation and memory. This immune system decomposition necessitates interfaces between MERCURY and the other systems responsible for analysis and elimination/repair. Coordination between these systems is required to ensure that the proper prevention strategies are used and that the proper response is taken following detection of the virus. The body uses chemical attractions and signals to coordinate the components of the immune system; the computer immune system must also coordinate its functions.

3.6. Summary

This chapter explained several analogies between a computer system and the human immune system by presenting several computer immune models. First, discussions of the human immune system and the public health system were presented, with research trends in the area of computer immunology addressed. The CHS was presented as an informal, explanatory model of the public health system; the computer immune system was presented as an informal, explanatory model of the human immune system. MERCURY's role in the computer immune system was defined. Chapters Four and Five provide general and detailed descriptions, respectively, of this virus detection component.

4. System Design

4.1. Overview

Chapter Two described current methods of virus detection, concepts of constructive induction, and research trends utilizing machine learning for computer security and detection. This chapter conjoins these areas to form the basis of MERCURY, the virus detection component of a computer immune system.

Chapter Three presented the Computer Health System, a global approach at virus prevention, based on the public health system. This approach identified the requirement for an individual computer immune system, based on the human immune system. Chapter Four focuses *only* on MERCURY and the design fundamentals of this virus detection system.

The methodology provides a framework for MERCURY by defining the objectives, requirements, and architecture of a *fully operational* detection system. The major subsystems and processes involved are decomposed and system integration and testing are addressed. The detailed descriptions of the subsystems and processes included in the *implemented prototype* of MERCURY will be provided in Chapter Five.

4.2. General Description

A system is a set of interrelated systems, or subsystems, working together toward some common objective [BlF90]. MERCURY is a virus detection system, composed of a virus scanner, a constructive induction based learning engine named HEC, and a knowledge base. This implementation is a proof-of-concept; HEC is a *model* of the learning process that provides the scanner with byte signatures that distinguish between self and nonself. This system tests the hypothesis that constructive induction can be effectively applied to the virus detection domain.

4.3. Objectives

An effective system engineering process begins by identifying a need, based on a *want* or *desire* for something, possibly arising from a deficiency [BlF90]. Current viral detection techniques are reactive, labor intensive for virus researchers, have a slow response from time of discovery until the cure is prescribed, and require user intervention to update the virus signature database [Kep94]. Due to these recognized inefficiencies, the need was identified for *an improved technique* that combats these virus detection problems.

Once the need is identified, system objectives must be explicitly defined and understood so that the system provides the desired output for each given set of inputs. The objectives of this detection system match the three most prominent functions the human immune system uses to defend the body against invasion. The objectives of

MERCURY are to: detect viruses, adapt to changes in self through learning, and remember previously seen viruses.

Table 15 -- MERCURY's Objectives

Detection	*The ability of the detection system to recognize nonself*
Adaptation	*The ability of the detection system to learn the structures of specific viruses; the detection system adapts to the changing definition of self*
Memory	*The ability of the detection system to efficiently and effectively remember previously seen viruses, speeding the response to subsequent encounters*

4.4. Requirements

Once system objectives are determined, *specific* requirements of the system must be defined. Requirements are decided prior to development so that expected system outputs are known and can be tested [BlF90]. A full implemented version of MERCURY should meet the following requirements:

1. Detect previously seen self in the system
2. Detect previously seen nonself in the system
3. Isolate and flag previously unseen files in the system
4. Send indiscernible files to virus experts
5. Relay virus information to the constructive induction engine for self-adaptation and learning
6. Update the knowledge base of virus signatures base on new concept of self
7. Send information about damaged files to the cleaner subsystem, outside of the MERCURY system

4.5. Architecture

After the system requirements are established, they are utilized to choose a design approach [BlF90]. Choosing a system architecture is a design approach that defines a system in terms of components, interactions and correspondence to system requirements [ShG96]. Implementing a particular architecture benefits the system's development and maintenance by defining the commonalties, identifying areas of reusability, and communicating the design to others. Architectures can be chosen by analyzing the applicability of different architectures and system requirements.

MERCURY's requirements involve collecting, manipulating, and preserving large bodies of complex virus information; overall, the system exhibits many qualities of a shared information system. This architecture supports independently processing subsystems interacting through a shared data store. When fully implemented, MERCURY subsystems would act independently; HEC and the virus scanner would each have processes that run autonomously within their respective subsystems. These subsystems would interact through a central knowledge base of virus information.

The definition of a system is not complete without considering its position in a larger, higher-level system. A virus detection system, such as MERCURY, would be a component of a individual computer immune system. This computer immune system is a component of the Computer Health System, designed and explained in Chapter Three. These systems protect computer systems against viruses through the detection, diagnoses, isolation, cure, and prevention of virus infections.

An individual computer immune system is responsible for system analysis, detection, adaptation, memory, and virus elimination/repair. MERCURY encompasses the functions of detection, adaptation, and memory, interfacing with the system analysis and virus elimination/repair components. Within MERCURY, there are three subsystems: virus scanner, constructive induction engine (HEC), and the knowledge base.. The simplified view of the Computer Health System, and MERCURY's position within it, are depicted in Figure 9.

4.6. Description of Subsystems

The following sections describe MERCURY's three subsystems, by providing their general descriptions, and defining their objectives, requirements, and architecture.

4.6.1. Constructive Induction Engine (HEC)

HEC is a software program developed as the constructive induction based learning engine. HEC is needed to drive the learning process for MERCURY, enabling the system to learn the definition of self and nonself and adapt to changes. The objectives of a constructive induction engine are to: generate, order, evaluate, and incorporate hypotheses [Gun91]. Hypothesis generation involves the creation of hypotheses based on predetermined rules and operators. Hypothesis ordering serves as a filter, identifying the most and least promising hypotheses. Hypothesis evaluation tests

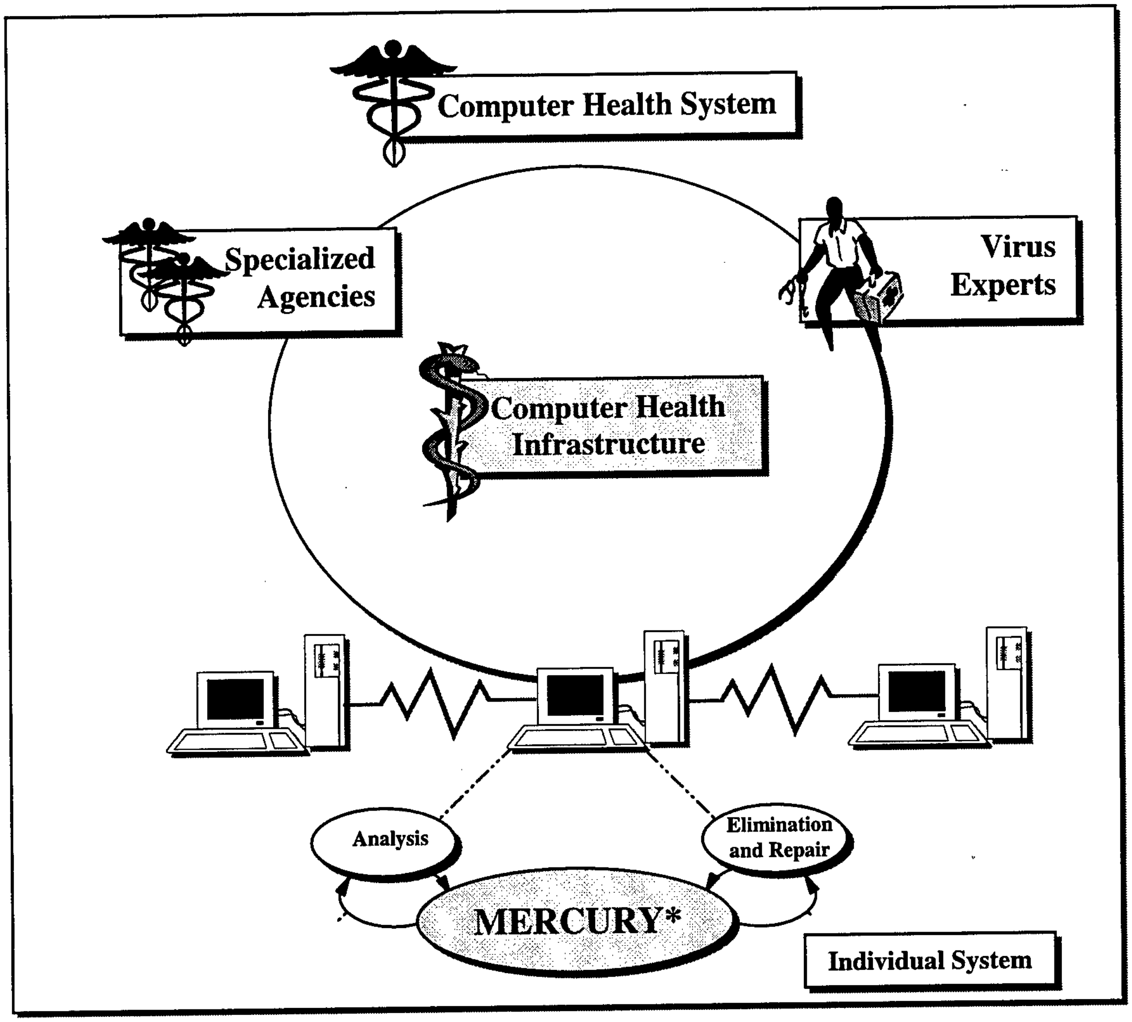

Figure 9 -- System Hierarchy

hypotheses to see if the goal of the system is achieved through their use. Hypothesis incorporation integrates the hypotheses determined to work properly into the rule base.

With these subsystem objectives determined, the *specific* requirements and expected outputs of a fully operational version of HEC are defined as follows:

1. Read-in a set of files identified as nonself or self
2. Use selective induction methods to generate all possible hypotheses from these files
3. Order and evaluate these generated hypotheses based on their abilities to distinguish between self and nonself
4. If generated hypotheses do not distinguish between self and nonself with a desired level of accuracy, use constructive induction methods to construct new hypotheses
5. Order and evaluate these constructed hypotheses based on their abilities to distinguish between self and nonself
6. Iterate through the constructive induction process until accepted level of accuracy is obtained
7. Output an accepted set of detectors, based on the hypotheses which best define the concepts of self and nonself

System architectures may be further refined as architectural subsystems. These subsystems are often developed independently, so they can be reused in different contexts [ShG96]. Similar to its "parent system," HEC exhibits many qualities of a shared information system since its main responsibilities are collecting, manipulating, and preserving large lists of *byte pattern* information. The processes of this subsystem that manipulate hypotheses are independent of each other, run in a fixed sequence until completion, and pass the hypothesis list to the next process for computation.

4.6.2. Virus Scanner

The virus scanner subsystem is needed to scan system files for viruses. The scanning method of virus detection was chosen over the other antivirus methods due to its ease of programming, modifiability, and function. Its simplicity is best suited for this proof-of-concept design to improve virus detection. The objectives of the virus scanner are to: scan system files for nonself, accept previously seen self, reject previously seen nonself, and flag previously unseen byte patterns, which could represent nonself.

With system objectives determined, *specific* requirements and expected outputs of a fully operational virus scanner, within MERCURY, are defined as follows:

1. Read byte patterns from system files
2. Access the byte patterns maintained in the knowledge base
3. Compare system byte patterns to those in the knowledge base
4. Accept files with self byte patterns from the knowledge base
5. Reject files with nonself byte patterns from the knowledge base
6. Question files with byte patterns not contained in the knowledge base
7. Send questionable files to the virus expert

The expected output from this subsystem is a decision to accept the file as self, reject the file as nonself, or flag the file as indiscernible, based on current knowledge.

The virus scanner follows the organization of a main program/subroutine architecture [ShG96]. The main program acts as the driver for the subroutines, providing a control loop for sequencing through the subroutines in a defined order. The virus scanner calls a read subroutine, which inputs all signatures from the knowledge base, then calls a locate subroutine, which finds all the files in system to be compared. Finally, it calls a compare subroutine, which compares the system file byte patterns to those in the signature list.

4.6.3. Knowledge Base

If fully implemented, MERCURY would utilize a knowledge base, or data repository that maintains all byte patterns used to define the concepts of self and nonself. This subsystem is needed to store and transfer information between the learning engine and the virus scanner. The objectives of the knowledge base are to maintain byte patterns, to accept data input from the constructive induction engine, and to provide signature access to the virus scanner. The *specific* requirements of the knowledge base are the ability to:

1. Maintain the data in the form of a signature list
2. Accept inputs from HEC in the form of a list of detectors
3. Provide read-only access to the virus scanner

The expected output from this subsystem is a database of signatures. The knowledge base does not generate any actions on its own, but instead responds to requests to store and access data.

4.7. Dynamic Structure of MERCURY

A system can be understood by examining its dynamic structure, which represents control information such as events, states, and operations occurring within a system. An event is a signal that something has happened. A state represents the system in the interval between events and specifies how events are interpreted. A transition between states represents the response to an event and may include actions and events to send to other system components. A transition may also contain guard conditions, which control whether a transition is allowed to occur. An action is an automatic operation in response to an event; one type is sending an event outside the system. A state diagram is a graph of states and transitions labeled by events. [RBP91]

The overall dynamic structure of a fully implemented version of MERCURY is depicted in the state diagram in Figure 10. The current prototype of MERCURY does not fully encapsulate this dynamic structure. This research was concerned with developing a proof-of-concept of the applicability of constructive induction to this domain. Therefore, development efforts focused on the inductive learning foundations of HEC.

The first state of MERCURY is "scanning the system for viruses." While scanning, if MERCURY encounters files with known byte patterns of self that are contained in the knowledge base, it continues. If MERCURY encounters files with at least one known nonself byte patterns it sends the "eliminate and repair message" to the elimination/repair system and continues scanning. If MERCURY locates a file with no byte patterns from the knowledge base, it sends the "identify file message" to the virus expert system and continues scanning. Once the virus expert has classified the file as self or nonself, MERCURY sends the classification and the file to the induction engine (HEC). MERCURY also sends the "eliminate and repair message" to the elimination/repair system if the file was identified as nonself. While the system continues to scan for other nonself files, HEC relearns the concept of self. It remains in this state until the learning process is complete and the new list of detectors is formed.

This research assumes that learning *will* be possible with the given methods of selective and constructive induction. There may be cases when learning is not possible, such as inconsistencies in the classification of examples. Once this process is complete, MERCURY enters the "update knowledge base" state of the system. During this state, scanning temporarily stops and the knowledge base accepts inputs from the induction engine, updating its signature list. Once the update is complete, MERCURY re-enters the "scanning the system for viruses" state and continues scanning for viruses, utilizing the new signature list.

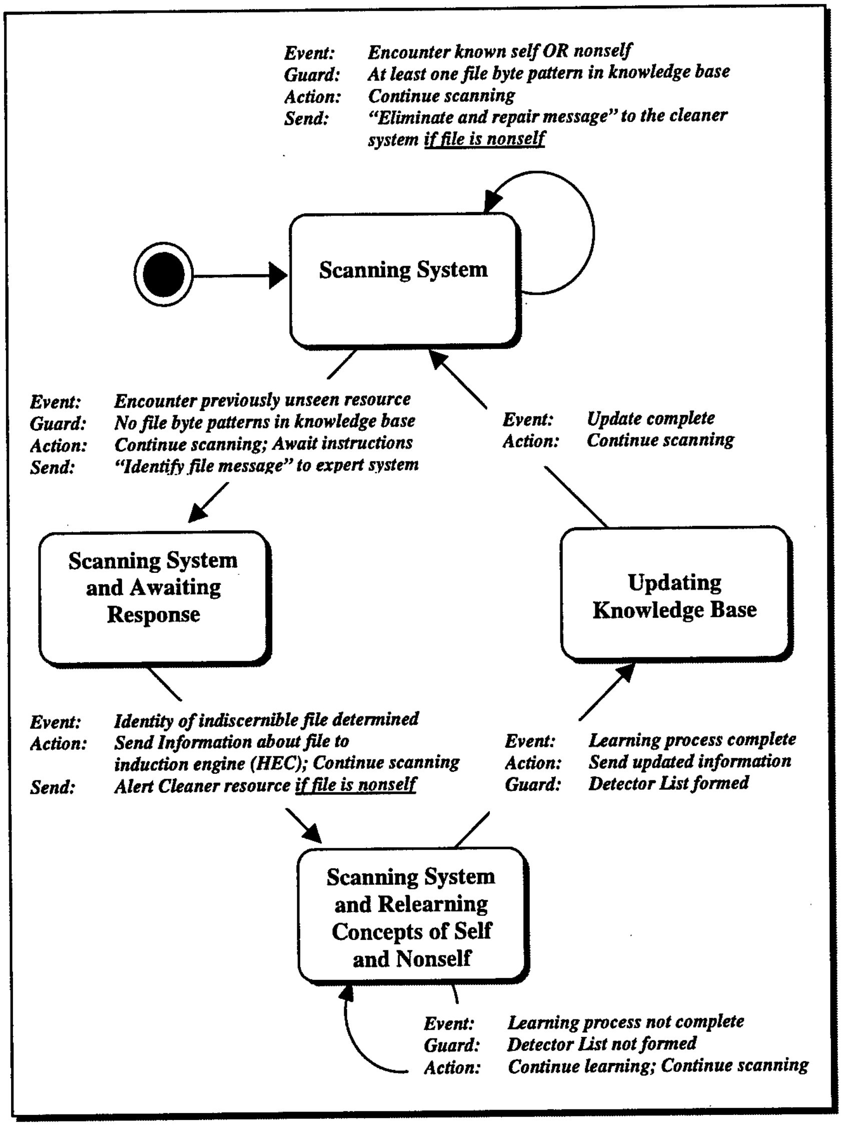
Event: Encounter known self OR nonself
Guard: At least one file byte pattern in knowledge base
Action: Continue scanning
Send: "Eliminate and repair message" to the cleaner system if file is nonself
Scanning System
Event: Encounter previously unseen resource
Guard: No file byte patterns in knowledge base
Action: Continue scanning; Await instructions
Send: "Identify file message" to expert system
Event: Update complete
Action: Continue scanning
Scanning System and Awaiting Response
Updating Knowledge Base
Event: Identity of indiscernible file determined
Action: Send Information about file to induction engine (HEC); Continue scanning
Send: Alert Cleaner resource if file is nonself
Event: Learning process complete
Action: Send updated information
Guard: Detector List formed
Scanning System and Relearning Concepts of Self and Nonself
Event: Learning process not complete
Guard: Detector List not formed
Action: Continue learning; Continue scanning

Figure 10 -- MERCURY's Dynamic Structure

To further understand the dynamic structure of MERCURY it is helpful to study the dynamic structure of MERCURY's main component, HEC. The dynamic structure of HEC is depicted in the state diagram in Figure 11. The current prototype of HEC does not fully encapsulate this dynamic structure. This research was concerned with developing a proof-of-concept of the applicability of constructive induction to this domain. This prototype is not optimized enough to be utilized in a fully operational system. Additionally, the "constructing detector list" state is not implemented.

The first state of HEC is "gathering examples." In this state, the engine reads in byte patterns from the provided example files identified as self or nonself. Next, HEC enters the "generating hypotheses based on selection methods" state, where it forms a list of hypotheses using predefined methods of selective induction. Once this hypothesis list is complete, the system begins "evaluating and ordering hypotheses." This state tests and ranks the ability of the hypotheses to distinguish between self and nonself, given as a score. This score, which measures the ability of the hypothesis to classify many examples correctly, is discussed in Section 5.3.2. If one or more hypotheses produce an acceptable score, the system generates a signature list, based on these hypotheses and enters the "constructing detector list" state. The system utilizes this list to update the knowledge base and redefine the concepts of self and nonself. If the hypothesis or hypothesis group *does not* produce an acceptable score, HEC enters the "formulating hypotheses based on construction" state, where it forms new hypotheses through constructive induction methods. These hypotheses are formed by combining previously generated hypotheses, using a predefined set of operators. After the combinations are complete, a constructed hypothesis list is passed back to the "evaluating and ordering hypotheses" state, where the score for each new hypothesis is tested and compared. HEC will continue to construct and evaluate hypotheses until it achieves acceptable score or until the system times out. All design considerations are discussed in Chapter Five.

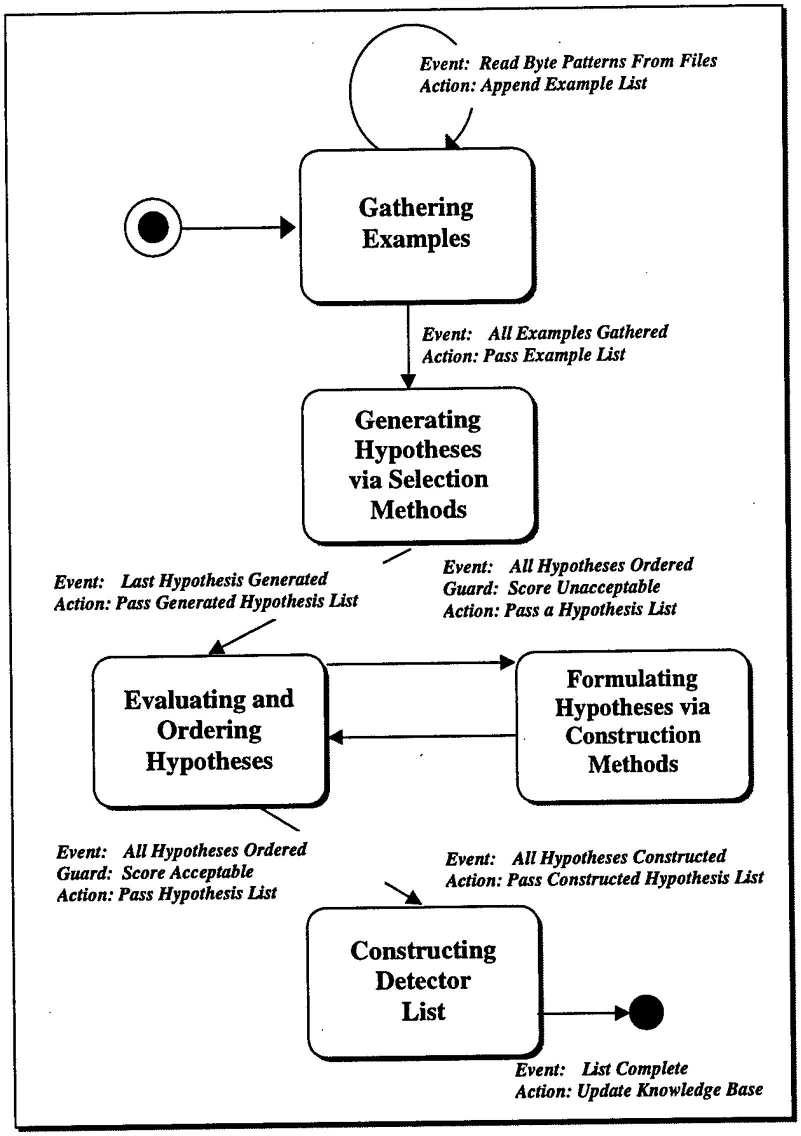
Event: Read Byte Patterns From Files
Action: Append Example List
Gathering Examples
Event: All Examples Gathered
Action: Pass Example List
Generating Hypotheses via Selection Methods
Event: Last Hypothesis Generated
Action: Pass Generated Hypothesis List
Event: All Hypotheses Ordered
Guard: Score Unacceptable
Action: Pass a Hypothesis List
Evaluating and Ordering Hypotheses
Formulating Hypotheses via Construction Methods
Event: All Hypotheses Ordered
Guard: Score Acceptable
Action: Pass Hypothesis List
Event: All Hypotheses Constructed
Action: Pass Constructed Hypothesis List
Constructing Detector List
Event: List Complete
Action: Update Knowledge Base

Figure 11 -- HEC's Dynamic Structure

4.8. Description of Data Flow

In addition to understanding the dynamic structure of MERCURY, it is also important to recognize the flow of data through a system. The data flow diagram in Figure 12 provides an overall view of the system by presenting the origination and destination of data and the processes that transform them [RBP91]. It shows the sequences of the transformations performed, as well as the external systems affecting the computation. Processes are depicted as circles, dataflows are depicted with arrows, and data stores are depicted with parallel lines. The current prototype of MERCURY does not fully implement this data flow. HEC does not fully interactive with the knowledge base and scanner, data flow is constrained within this system. Additionally, this design includes an incremental learning mechanism, which has not been incorporated into HEC. An operational version would complete these interactions.

MERCURY's first process scans the system by comparing known byte patterns from the knowledge base to byte patterns from files in the system. The next process determines if an unclassified file is self or nonself, through a virus expert system. Once the file has been identified as self or nonself, the file and the virus expert's classification are transferred to HEC. This adaptive component will induce a detector for the file that is consistent with the existing definitions of self and nonself. Upon completion of this process, the new detector is transferred to the knowledge base, in the form of byte patterns.

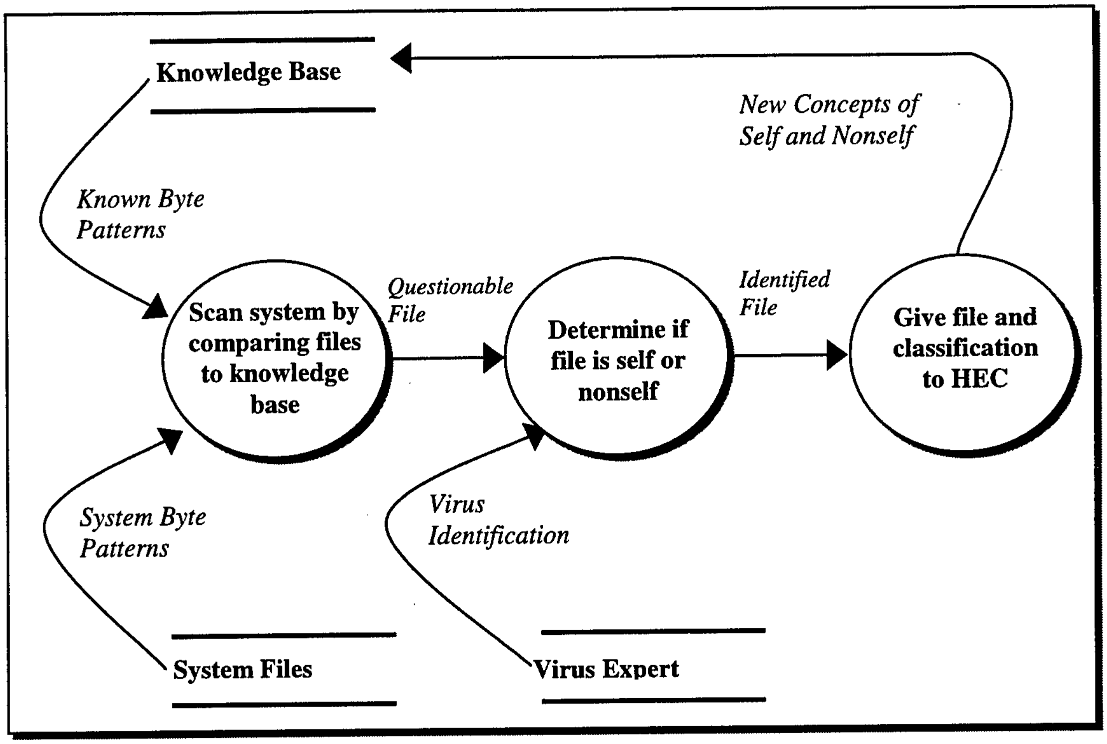
Knowledge Base
New Concepts of Self and Nonself
Known Byte Patterns
Scan system by comparing files to knowledge base
Questionable File
Determine if file is self or nonself
Identified File
Give file and classification to HEC
System Byte Patterns
Virus Identification
System Files
Virus Expert

Figure 12 -- MERCURY's Dataflow Diagram

4.9. System Integration and Testing

The high-level integration of MERCURY involves matching the inputs to outputs between the major subsystems: HEC, virus scanner, and knowledge base. This integration step is necessary for the operational version of MERCURY. The virus scanner must be able to access the byte patterns in the knowledge base. The virus scanner and the inductive engine must be to communicate through the virus expert and the knowledge base. Figure 13 loosely describes the exchange of data between the subsystems, depicted as black boxes, with notional inputs and outputs.

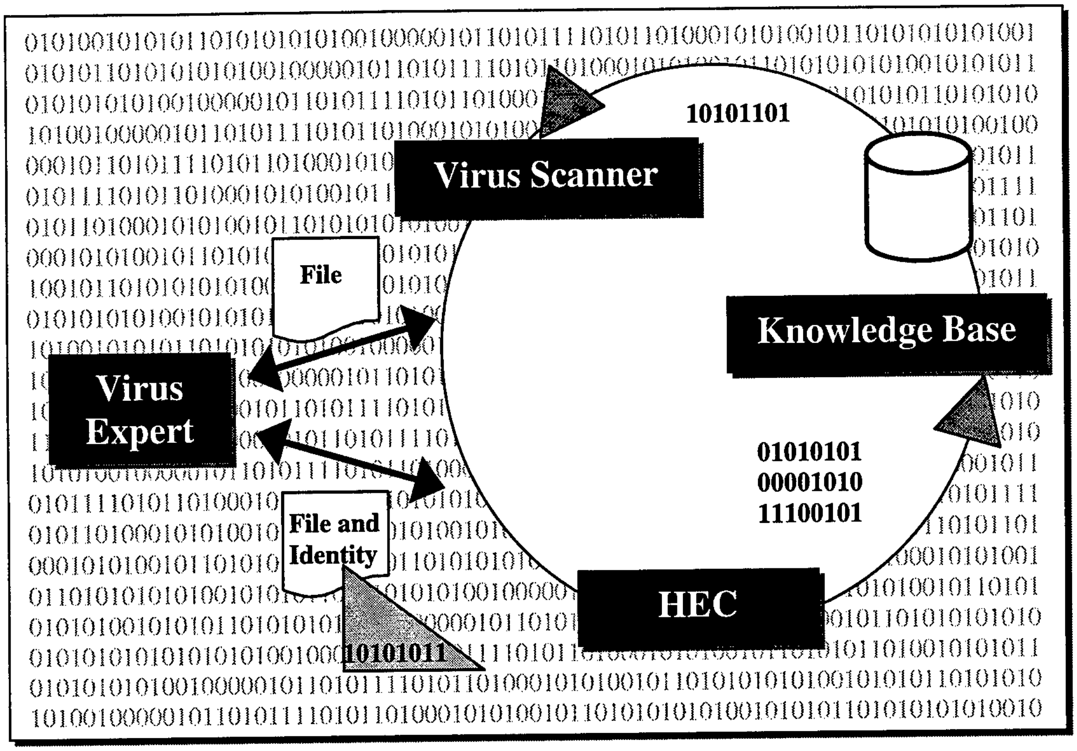

Figure 13 -- MERCURY's Integration

Testing of the fully operational version of MERCURY should occur in several phases. First, each of the subsystems should be tested as independent units, to assure individual requirements were met. Once each unit is determined to function properly, each of the three interfaces should be tested: HEC's interface with the knowledge base, the virus scanner's interface with knowledge base, and virus scanner's interface with HEC, through the virus expert. After this testing is complete, the entire system should be tested. While testing the system, the following should be considered:

1. Overall system objectives and requirements were met
2. Subsystem objectives and requirements were met
3. System and subsystem outputs are valid
4. System and subsystem processes operate accurately and efficiently
5. Data integrity was maintained throughout the system

4.10. Summary

The Computer Health System, presented in Chapter Three, identified the need for an individual computer immune system. This chapter focused on its main component, MERCURY. The design presented here provided a high-level description of an operational version of MERCURY, by defining its objectives, requirements, and architecture, the major subsystems and processes involved, and their integration and testing. Deviations between this *proposed* design and the *prototyped* implementation were noted.

An important aspect addressed in this chapter was the dynamic structure of MERCURY, detailing the system states and the sequences of actions and events with trigger transitions. Within MERCURY, there is a constant flow of data between the subsystems. This chapter discussed data origination and destination, as well as the integration required to enable transmission between subsystems. Finally, the levels of testing were briefly discussed. Chapter Six provides specific information about the types of tests conducted with the prototype.

The detailed descriptions of design decisions, tradeoffs, and limitations for each subsystem and process within MERCURY are provided in Chapter Five. Chapter Six discusses the testing methods utilized to validate and verify the output of MERCURY and provides the results of the testing.

5. System Implementation

5.1. Overview

Chapter Four presented the high-level description of an operational version of MERCURY. This chapter highlights the *prototyped* implementation of this *proposed* design. This chapter provides a detailed description of the components and processes within the *prototyped* version of MERCURY, including the *prototyped* versions of its three main components: the constructive engine, the scanner, and the knowledge base. Each of these components is represented abstractly, not practically, in the system. The system components are explained in terms of their functions, design decisions and corresponding advantages and disadvantages, tradeoffs and limitations, complexity, and modifiability. Chapter Six presents the results of the testing of MERCURY. Chapter Seven presents the conclusions of this research.

5.2. Definitions

Several terms will be used extensively in the discussion of the detailed design. This section clarifies their meaning in relation to the detection system:

Signature A series of bytes from a file that distinguishes self for nonself.

Hypothesis A candidate signature formed by HEC. A hypothesis is composed of a label, features, the generation method, and a score that indicates the "goodness" of the hypothesis.

Example A labeled instance of the concept to be classified. In this system an example is a file that is labeled as self or nonself.

Label A flag that indicates whether the example is an instance of self or nonself. The label is also used to flag whether the hypothesis classifies self or nonself.

Attribute A byte from a file. A byte is composed of 8 bits, represented by logical 1s and 0s.

Feature A sequence of 16 attributes that can be used to identify a file as self or nonself. The absolute position of the attributes is not considered, although relative position is used by the selection rules. A hypothesis generated by construction contains multiple features, while a hypothesis generated by selection contains one feature.

Generation Method The series of steps that were used to create the features.

Concept An abstraction that a file belongs to a class. The concept space of this research is self and nonself.

5.3. Construction Induction Engine

HEC develops signatures for both self and nonself files. Self signatures are used to classify files currently on the computer. Nonself signatures store information about previously seen viruses. The use of self and nonself detectors allow for extensibility within the computer immune system by allowing the ability to share virus information across computer systems. This capability is analogous to a vaccination shared among a population, in order to prevent the spread of disease.

The constructive induction engine is responsible for providing MERCURY's virus scanner the signatures needed to distinguish self from nonself. HEC performs the four constructive induction processes that were discussed in the literature review: hypothesis generation, evaluation, ordering, and incorporation. The following sections cover the implementation of these processes and the required design decisions.

5.3.1. Hypothesis Generation

Hypothesis generation involves the creation of hypotheses based on predetermined rules and operators. HEC creates hypotheses based on two methods: initial *selection* of attributes from an example file, or *construction* based upon the features of two existing hypotheses from the same concept. HEC creates hypotheses by selective, then constructive induction. The following subsections describe the notion of a hypothesis, explain the selection rules and define the constructive operators and generation grammar.

5.3.1.1. Hypothesis

A hypothesis is a candidate solution to the problem the system was assigned to solve; however, more than one hypothesis may be needed to solve a problem. Inductive systems are based upon the creation, manipulation and evaluation of hypotheses. Hypotheses in HEC represent a candidate signature, or a sequence of bytes that identify a file as either self or nonself.

In HEC, a hypothesis is composed of a label, feature, generation method, and score for effectiveness. The label indicates whether the hypothesis classifies self or nonself. During creation of a hypothesis by a selection rule, the hypothesis' label is given the same value as the example. A feature is a sequence of 16 bytes that may uniquely identify the example. This size was chosen based upon research done at IBM. This research empirically determined that a 16 byte signature is sufficient to provide a high detection rate with a false positive probability of less than 0.5% [KeA94]. Additionally, 16 bytes is the signature size used in current virus research, as noted in the industry publication *Virus Bulletin* [KSSW97]. The generation method explains how the hypothesis was created; this information is needed during evaluation and by the scanner. The score for effectiveness measures how effectively the hypothesis distinguishes between self and nonself. This score is discussed in the hypothesis evaluation subsection. Hypotheses are stored as a list of records. The list data structure was chosen for ease of manipulation. The record for each hypothesis contains three fields, *label*, *method*, and *score*.

This research hypothesized that a series of bytes can be used as an effective inductive feature to distinguish between self and nonself in a file. Other features could be used for this purpose, notably system calls or system state. Different antivirus programs have been created that use these features to determine the existence of viruses in a computer. MERCURY utilizes a scanner type antivirus program, since this type effectively recognizes byte signatures in files.

5.3.1.2. Example Set

The example set is a collection of files that are labeled as instances of self or nonself. Through the inductive process, HEC seeks to create signatures which classify members of the example set as self or nonself. Some researchers use the term training set in lieu of example set, since the examples are used to train the system in the problem area.

The example set is specified by a file that contains a listing of the file names and associated self or nonself labels. Self and nonself examples are included in the example set to ensure that signatures for each concept are induced from examples. Another technique would include examples from one concept only, allowing the system to detect members of the other concept as deviations. A system that trains on one concept would need to learn the most general hypothesis, while a system that trains on all possible concepts can learn a more specific hypothesis [Hau87].

5.3.1.3. Selection Rules

HEC begins as a *selective* induction system; it *only* selects existing attributes to describe the concept. Any rule that extracts bytes from the example can be used for selection. Possible selection rules include: selecting bytes from a certain portion of a file, selecting random bytes from a file, selecting certain chunks from a file, selecting bytes through a sliding window moved across the file, or selecting every Nth byte.

Based upon storage and computational limitations and constrained by the scope of this research, selection rules were limited. HEC uses three selection rules, *chunking*, *sliding window*, and *every other byte sliding window.* Table 16 explains these rules.

Table 16 -- Selection rules

Rule	Description	# Hypotheses Created[1]
Chunking Selection	This rule breaks an example into non-overlapping segments containing an equal number of bytes.	N/K
Sliding Window Selection	This rule divides an example into overlapping segments containing an equal number of bytes.	N – K + 1
Every Other Byte Sliding Window Selection	This rule divides an example into overlapping segments, extracting every other byte from each segment until the specified number of bytes are selected.	N – (2K – 1)

By way of illustration, Figure 14 shows how the selection rules create hypotheses from the examples.

[1] N is the number of bytes in the example. K is the number of bytes in each feature field of the hypothesis.

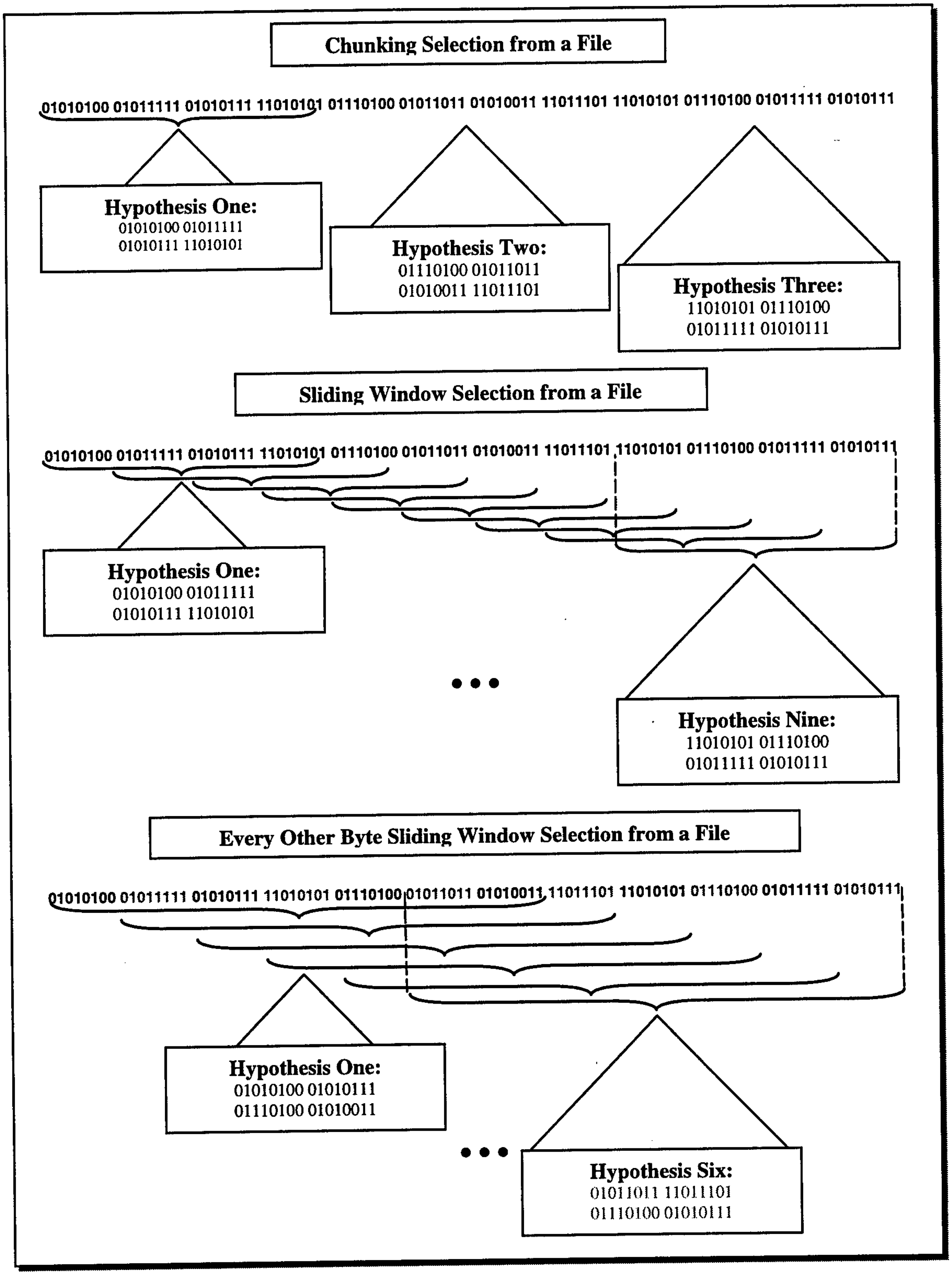

Figure 14 -- Selection rules

5.3.1.3.1. Bias

The choice of selection rules is a form of bias since these rules are based upon background knowledge gained from other antivirus research. *Sliding window* selection is similar to the current method of choosing byte signatures for a scanner based antivirus program. These methods try to find a series of bytes in a file that can be found anywhere in the file. *Chunking* selection was included in this research since this rule produces features that are a subset of those produced by sliding window selection. *Every other byte sliding window (EOSW)* selection was included in this set of selection rules to increase the probability of capturing patterns across a greater number of bytes. This form of selection *could be* interpreted as a form of construction, since these selection rules combine attributes through conjunction, creating a new feature.

5.3.1.3.2. Process / Algorithm

```
HypoList = []
Read in file and associated label
Use chunking selection rule to create hypotheses from file
Use sliding window selection rule to create hypotheses from file
Use every other byte sliding window selection rule to create
       hypotheses from file
HypoList = hypotheses from chunking, sliding window and every
       other byte sliding window
```

Figure 15 -- Algorithm for Selecting Hypotheses

During the first hypotheses generation process, files from the example set are read by HEC. Each selection rule is applied to these files, resulting in a list of hypotheses.

5.3.1.3.3. Computational Complexity

Computational complexity captures the order of growth of an algorithm and gives a simple characterization of the algorithm's efficiency. This measure allows computer scientists to compare the relative performance of algorithms, identify areas of improvement, and highlight expected pitfalls during execution [CLR90].

Each selection rule is applied to each file from the example set. The number of hypotheses selected is the important factor in the computational complexity of selection. Based upon Table 16, the computational complexity of selection is *O(n-k)*, or linear.

5.3.1.4. Hypothesis Construction

The other method of hypothesis generation in HEC is hypothesis construction. Hypotheses are constructed by combining existing hypotheses through constructive operators. This constructive process can be repeated if "better" hypotheses are needed. Initially, hypotheses are constructed solely from selected hypotheses, although constructed hypotheses can be used as input for later iterations of construction.

5.3.1.4.1. Constructive Operators

Constructive operators create new hypotheses by specifying relationships among features of existing hypotheses. These operators can manipulate the features through logic, mathematics, statistics, heuristics, or a variety of other means.

HEC uses three logical operators and two spatial operators. The logical operators perform a Boolean comparison of the existence of two features. The three logical operators are AND, OR, and XOR. These operators input two hypotheses with the same label and the same selection method. The constructive process outputs a single hypothesis that captures the essence of: both features in a file, one feature or another or both are in a file, or one feature or another has to be in the file, but not both simultaneously. Given N hypotheses, each logical operator will construct C(N,2) or N choose 2, hypotheses. Figures 16 through 18 illustrate the logical operators constructing a new hypothesis from two existing hypotheses.

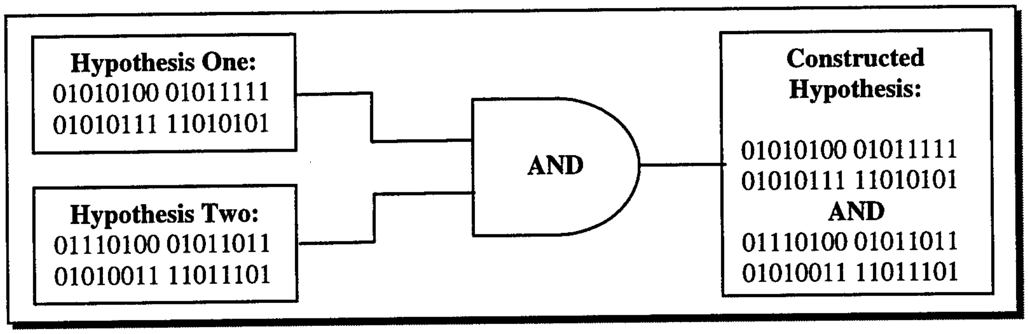

Figure 16 -- Pictorial Representation of the AND Operator

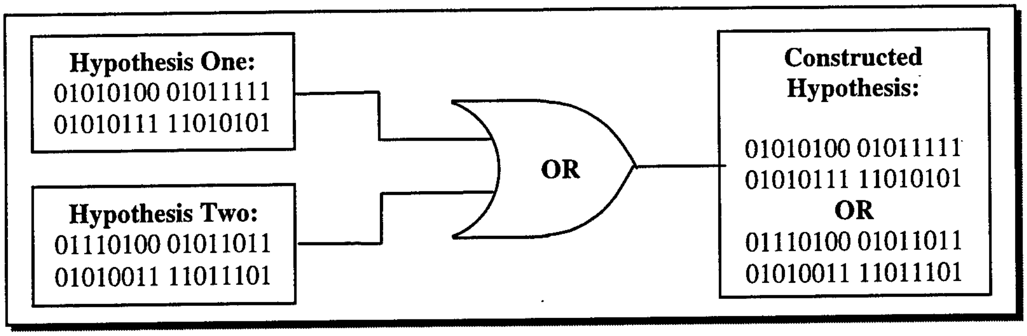

Figure 17 -- Pictorial Representation of the OR Operator

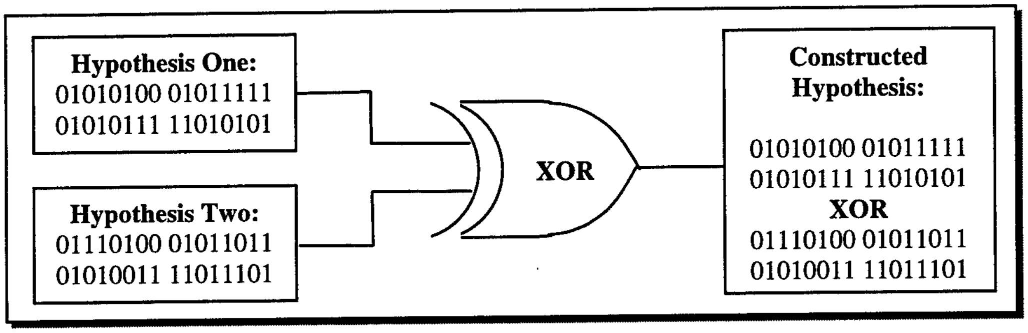

Figure 18 -- Pictorial Representation of the XOR Operator

The spatial operators compare the relation of the features' relative locations in a file. The two spatial operators are BEFORE and DISTANCE. These operators input two hypotheses with the same label, the same selection method, and created from the same example file. The operators output a single hypothesis that captures the idea that one feature is before the other feature, or that one feature must be a certain number of attributes, or bytes, away from the other feature.

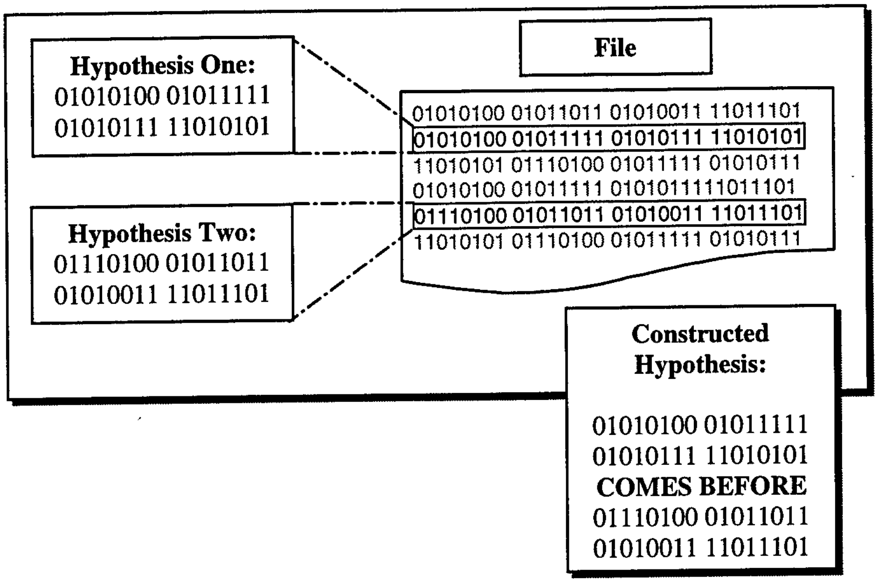

Figure 19 -- Pictorial Representation of the BEFORE Operator

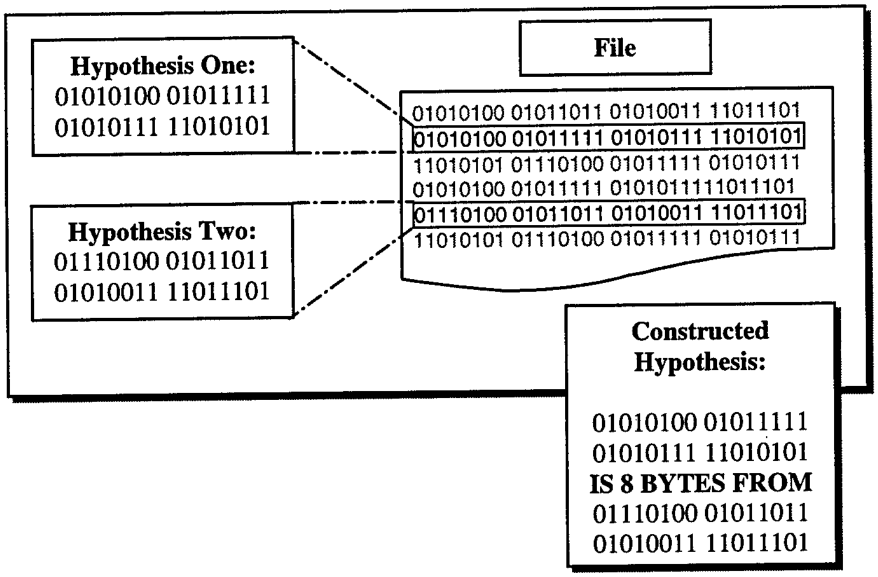

Figure 20 -- Pictorial Representation of the DISTANCE Operator

The distance between features is based upon the number of bytes between the last feature of the first hypothesis and the first feature of the second hypothesis. This method does not account for hypotheses with interleaved features. Given N hypotheses, each spatial operator will construct C(N,2) or N choose 2, hypotheses. Figures 19 and 20 illustrate the relation between the new constructed hypothesis and the original hypotheses from which it was formed.

5.3.1.4.2. Bias

Several biases were used in determining whether to apply a constructive operator to two hypotheses. The first bias only allows construction on hypotheses from the same concept as determined by the label. This bias ensures that the hypothesis' label is not assigned randomly; rather, the label is based upon previous information.

Another bias limits construction to existing features, instead of allowing randomly generated features. During the selection process, HEC extracts all possible features from the file based upon the selection rules. Randomly generated features are not guaranteed to be found in the example files. Randomly generated features that are found in the example files are duplicates of existing selected features. Since randomly generated features would not provide any new information or insight, they are not used in HEC. From a system perspective, scanning for randomly located byte patterns is unnecessarily complex, compared to a methodical approach.

Another bias limits the application of operators to hypotheses created through the same selection rule. This bias was included in the system to limit the different methods of file reading the virus scanner would need to use when searching for a detector in a file. This bias also reduces the number of constructed hypotheses. Finally, removing the possibility of combining selection rules in a hypothesis aids in isolating and analyzing whether a particular constructive operator improves the performance of HEC. The main objective of this research was to demonstrate the applicability of constructive induction to virus detection; therefore, this bias simplified the testing of this objective. It simplified testing by concentrating attention on the effect of operators used for construction, rather than the effects of combining selection rules.

The choice of constructive operators is based upon background knowledge of the binary structure of executable programs and virus research. The logical operators account for multiple, or restricted virus characteristics. Logical operators also account for common information that can be found in uninfected programs. The spatial operators account for the positioning of virus characteristics and nonvirus characteristics in an infected file.

Bias also restricted the application of the operators to two hypotheses, based upon the example file from which the hypotheses were derived. The first operator bias involves the AND operator. The AND operator is only applied to hypotheses with features from the same file. Since features are derived from an example file, the selected hypotheses are guaranteed to detect at least one example correctly. Without this bias, it is possible to create hypotheses using AND that do not classify any examples, reducing system performance when compared to selection. Another bias involves the XOR operator. Since XOR captures the essence that one feature or another feature is found in the file, but not both, a check is made prior to construction to ensure the hypotheses do not contain features generated from the same file. Other biases involve the spatial operators. The spatial operators BEFORE and DISTANCE are only applied to hypotheses derived from the same example file. This guarantees that the constructor will be able to determine a distance and ordering for the features in the file. The final operator bias is for the OR operator, which has no restrictions. The use of these biases reduces the overall number of hypotheses generated through construction.

A variety of techniques can be used in the application of constructive operators across the hypotheses. One technique would apply all the operators across two hypotheses and then evaluate the results. Another technique would apply an operator across all the hypotheses, evaluate the results from that operator, and if necessary, apply the next operator in succession. A third technique would apply all the operators across all hypotheses and then perform hypothesis evaluation. HEC utilizes the third technique. This decision was based on the need to test this research's primary hypothesis that constructive induction provides a suitable learning mechanism for the virus detector system of a computer immune system. Each operator is applied to all the hypotheses to see if construction improves the overall detection potential of the system. This technique also tests the constructive operator hypothesis by providing information on whether the spatial and logical operators can be used to construct effective hypotheses. Finally, since constructive induction has not been used in previous virus detectors, no background knowledge exists that can guide which operators will lead to the best detector.

5.3.1.4.3. Generation Grammar

A hypothesis can be created in several ways through the selection rules and constructive operators. Information is stored in the generation method field of the hypothesis that records the selection rule used to extract the features from the file and the relationship between the features established by the constructive operators. Evaluation of the hypothesis is accomplished by comparing the hypothesis to the examples and scoring the hypothesis. During this comparison, the features are found using the selection method and the relation between the features is compared to the generation grammar to determine if the hypothesis classifies the example.

The generation method field is instantiated through the generation grammar. This grammar is similar to prefix mathematical notation, where the selection rules are analogous to operands and constructive operators are similar to mathematical operators. The productions for the generation grammar are:

S → R

S → OSS

R → chunking | sliding_window | eo_sliding_window

O → and | or | xor | before | distance

By way of example, the generation method stored for a hypothesis created by applying the OR operator to two features selected through chunking rule would be [or, chunking, chunking]. The generation method is stored in a binary tree data structure inside the method field of the hypothesis.

5.3.1.4.4. Process / Algorithm

During constructive induction, the various constructive operators are applied across all the hypotheses. For each operator, two hypotheses are chosen from the hypothesis list, and the labels and selection *method* are checked to ensure they match. The new hypothesis is created by setting the *label* equal to the label of the original hypotheses and the *score* is set to null. The algorithm for this process is shown in Figure 21.

The next steps are dependent on the type of operator being used. Construction through logical operators is accomplished by ensuring that any biases applicable to the operator are satisfied. Next, the *method* is constructed by placing the operator at the root of the new expression, with the left and right methods from the original hypotheses. Construction through spatial operators is accomplished by ensuring that the hypotheses are from the same file. Following this step, the file from which the hypotheses were selected is searched to determine the relative location of the features for BEFORE and the number of bytes between the features for DISTANCE. This process is continued until each constructive operator has been applied to the hypotheses. This method does not exhaustively search for repeated features.

```
ConstructedList = []
HypoList = hypotheses to be used for construction Loop
Loop
     Hypo1 = first hypothesis in HypoList
     Tail = remainder of HypoList after Hypo1
     Loop
          Hypo2 = first hypothesis from Tail such that
                   Hypo2.Label = Hypo1.Label and selection rule for
                   Hypo1 and Hypo2 are the same
          Apply constructive operators to Hypo1 and Hypo2, storing
                new hypothesis in ConstructedList
     End Loop when Tail = []
End Loop when HypoList = []
ConstructedList = Hypotheses created from applying constructive
       operators to hypotheses in HypoList
```

Figure 21 -- Algorithm for Constructing Hypotheses

5.3.1.4.5. Computational Complexity

Constructing hypotheses uses a doubly nested loop to select the hypotheses two at a time in a combinatorial fashion. The complexity of this loop is based upon the number of hypotheses in the *HypoList*, or *n*. The application of the constructive operator takes constant time for logical operators and is based upon the size of the file, or *f*, for spatial operators in order to perform the search for the features. Based upon this analysis, the algorithm for constructing hypotheses takes $O(n^2f)$.

5.3.2. Hypothesis Evaluation

Hypothesis evaluation is responsible for determining a score of “goodness” for each hypothesis. This score is used to determine if the hypothesis should remain in the system for possible construction or incorporation into the knowledge base. During hypothesis evaluation, each hypothesis is compared against the example set to determine the number of examples that the hypothesis classified and whether these classifications were correct.

5.3.2.1. Scoring

MERCURY classifies files as self or nonself based upon virus signatures learned by HEC. HEC is responsible for learning signatures for self and nonself that are able to accurately classify files and have a predictive capability. This capability determines whether a previously unseen file is similar to existing self or nonself. In order to achieve a globally maximum accuracy and predictive capability, HEC seeks to induce hypotheses that are locally maximal. The hypothesis score is used to determine if the hypothesis is locally maximal.

Classification is when the features of the hypothesis are found in the example and this match of features satisfies the expression stored in the generation *method* field of the hypothesis. Misclassification is a form of classification where the labels of the example and the hypothesis are different. Correct classification is when the hypothesis classifies the example and the labels of the hypothesis and the example are the same.

A variety of measures can determine the effectiveness of a hypothesis for classifying files as self or nonself. Many existing constructive induction systems have used information gain for this purpose. Information gain is a measure of how well an attribute splits the examples into groups; i.e. the reduction of system entropy by the hypothesis. Information gain is computed through the following set of equations, where p is the number of examples classified in class P, n is the number of examples classified in class N, and v is the total number of examples.

$$I(p,n) = -(\frac{p}{p+n}\log_2\frac{p}{p+n}) - (\frac{n}{p+n}\log_2\frac{n}{p+n})$$

$$E(A) = \sum_{i=1}^{v}\frac{p_i+n_i}{p+n}I(p_i,n_i)$$

$$gain(A) = I(p,n) - E(A)$$

I(p,n) captures the idea that the probability of any example belonging to class P is p/(p+n) and the probability of any example belonging to class N is n/(p+n). E(A) is the expected information requirement for examples that have a particular attribute, A. The information gain is then the amount of entropy reduction ascribed to a particular attribute. [Qui86]

In HEC, the hypothesis is labeled during generation. During evaluation, HEC must determine how well the hypothesis detects examples with the same label. Using information gain in HEC would result in assigning the same score to hypotheses that classify many examples correctly and to hypotheses that classify many examples incorrectly. This property of information gain is based upon the score's normal usage, where the system assigns a label to the classifier after evaluation. This property is based upon the log_2 terms used in the calculation for I(p,n). Based upon this property, information gain is not used in HEC. Instead, other measurement criteria are used to evaluate hypotheses.

The hypothesis score used in HEC reflects three important measurement criteria of effectiveness: power, purity, and complexity. Power indicates the capability of the hypothesis to classify *any* given example. Purity reflects the ability of the system to be correct when it classifies. Complexity is related to Occam's razor; simple hypotheses should be preferred over those that are complex. Based upon these measurement criteria, the measurements of Table 17 are possible. In order to illustrate how these measures are calculated, an example is provided in Figure 23.

Table 17 -- Possible Measurements for Hypothesis Effectiveness

Measurement One	
Power	This measurement calculates the strength of the hypothesis by rating the percentage of classified examples over the entire set of examples. This measure does not account for correctness.
Measurement Two	
Purity	This measurement examines the group classified as belonging to a specific concept; in order to calculate the percentage of examples classified correctly over the total number of examples classified by that hypothesis.
Measurement Three	
Complexity	This measurement is based on Occman's Razor [Gun91], which states "Pluralitas non est ponenda sine neccesitat," meaning entities should not be multiplied unnecessarily. This measure is calculated by looking at the depth of the tree stored in the *method* field of the hypothesis, with a complexity of 0 for selection.

A desired measure is one that rewards a hypothesis for low complexity, high power and high purity. This concept is depicted in Figure 22. Hypotheses that posses this desired property are locally maximal and encourage the system to become globally maximal.

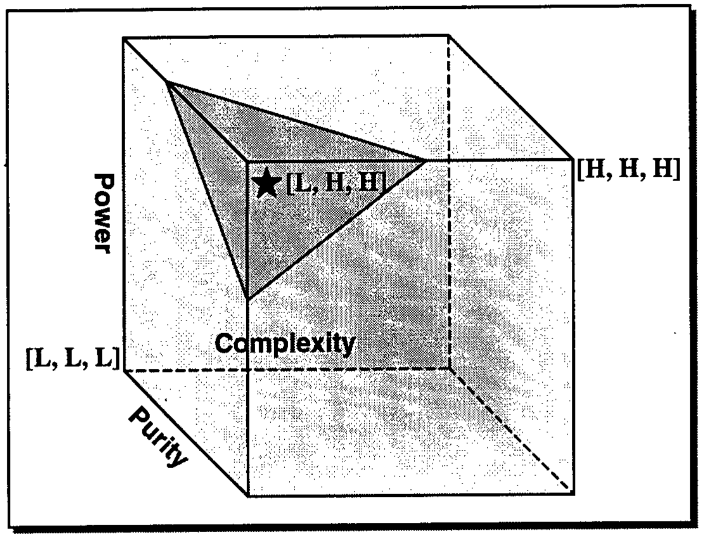

Figure 22 -- Depiction of Optimal Scores

The score for each hypothesis is stored as a record with fields for complexity, power and purity. These measurements require the evaluation process to determine the following metrics for each hypothesis: *R*, the number of examples classified correctly; *S*, the number of self examples; *N*, the number of nonself examples; *C*, the number of examples classified; and *D*, the number of constructions.

Based on this information, the score can be calculated by:

$$\begin{aligned} \mathit{Power} &= \frac{C}{N+S} \\ \mathit{Purity} &= \frac{R}{C} \\ \mathit{NumConstru\ ctions} &= D \end{aligned}$$

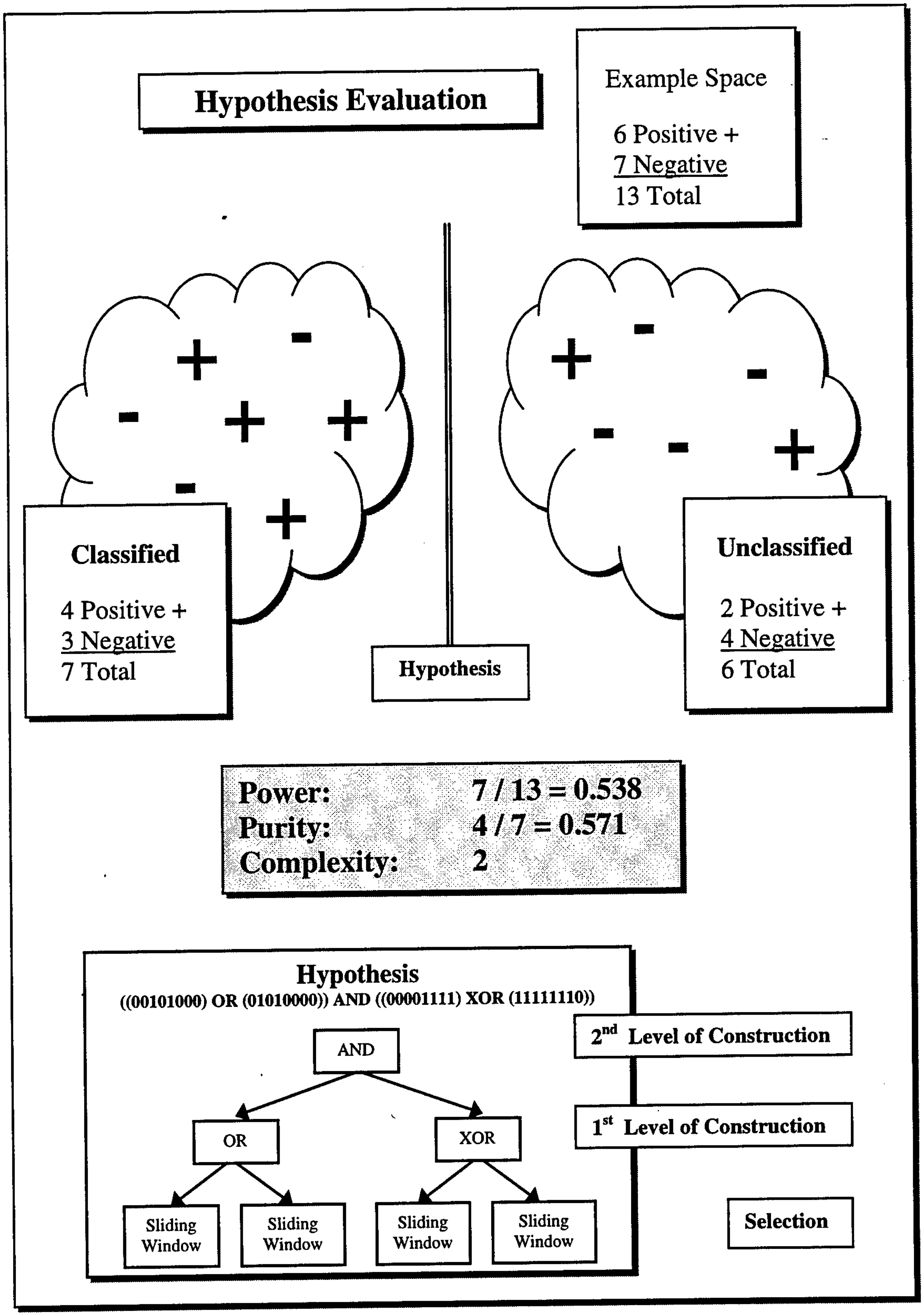

Figure 23 -- Example of Calculated Scores

5.3.2.2. Coverage

Power, purity and number of constructions provide a local perspective on the performance of an individual hypothesis. A global perspective is also needed to ensure the hypothesis list, as a whole, can classify the example set. This perspective is encapsulated by the concept of coverage. Coverage is derived by comparing the hypotheses to the examples to determine the number of hypotheses that classify, misclassify and fail to classify each example. Coverage shows whether the hypotheses are able to classify the entire example set, or only a certain portion. This measure is based on the percentage of examples classified by the set of hypotheses. Coverage is calculated after evaluation in a separate procedure.

5.3.2.3. Process / Algorithm

```
R,S,N,C <- 0
HypoList = hypotheses to be evaluated
ExampleList = examples that hypotheses are evaluated against
Loop
    Hypo = first hypothesis in HypoList
    Loop
        Example = first example in ExampleList
        If Example.Label = Self
            S <- S + 1
        Else
            N <- N + 1
        End If
        P <- Parse(Hypo, Example)
        If P = Found
            C <- C + 1
            If Hypo.Label = Example.Label
                R <- R + 1
            End If
        End If
    End Loop when ExampleList = []
    Hypo.Score = Score(Hypo)
End Loop when all hypotheses are evaluated
```

Figure 24 -- Algorithm for Hypothesis Evaluation

During the evaluation algorithm, depicted in Figure 24, each hypothesis is compared to all examples to determine its score. The *parse* function is used to compare a hypothesis from the list of hypotheses and an example chosen from the example list. Parse is responsible for determining whether the *features* were found and the *method* was satisfied. If the features were found and method was satisfied, the hypothesis classifies the example. This classification is correct if the labels of the hypothesis and the example match, otherwise this hypothesis misclassifies the example. *Score* is responsible for performing the calculations for the hypothesis score and storing this information in the hypothesis. The evaluation process stops when all the hypotheses have been compared against all the examples.

5.3.2.4. Computational Complexity

Hypothesis evaluation is accomplished through a doubly nested loop. The first loop traverses the hypothesis list, one at a time. This loop is executed *n* times, where *n* is the number of hypotheses in the list. The second loop iterates through the list of examples *e* times, where *e* is the number of examples. Therefore, evaluation has computational complexity of $O(en)$, or $O(n^2)$ in the worst case.

5.3.3. Hypothesis Ordering

The hypothesis ordering step is performed before or after the evaluation step, with the purpose of reducing the number of hypotheses needed for evaluation or construction, respectively. HEC has the ability to order hypothesis after evaluation, increasing the efficiency of the system. Bias or other heuristics can be used prior to evaluation to reduce the number of hypotheses that need to undergo this computationally complex operation. After evaluation, hypotheses that do not perform well can be removed in a manner similar to genetic algorithms or beam search.

HEC orders the hypotheses after evaluation, with the purpose of removing poor performers from the pool of hypotheses sent to construction. The decision to evaluate before ordering is based on the research objectives of determining what constructive induction components are viable in this domain. Ordering is used to improve the average case performance of construction, which is an inherently combinatorial operation. Ordering searches for hypotheses that are not dominated by other hypotheses. These hypotheses have the highest score for power and purity. The following sections examine the growth in the number of hypotheses without ordering, changes to the evaluation process, and ordering the hypotheses to find the nondominated set.

5.3.3.1. Hypothesis Growth

HEC generates numerous hypotheses in order to find the set of hypotheses that adequately classifies the examples. This subsection explores the growth of hypotheses based upon the various generation methods. Using the selection rules, selection creates at most

$$N/K + (N\text{-}K\text{+}1)\text{+ } (N-(2K-1))$$

which by reduction, is equivalent to

$$N/K\text{+}2N\text{-}3K\text{+}2$$

hypotheses from a file, where N is the number of bytes in the file and K is the number of attributes in each feature.

Following selection, hypotheses are used for construction. Construction chooses two hypotheses at a time, in a combinatorial fashion, and applies the five constructive operators. This process can be examined in terms of the number of hypotheses sent to construction, or *x*.

$$5\begin{pmatrix} x \\ 2 \end{pmatrix} = 5(\frac{x(x-1)}{2}) = \frac{5}{2}x^2 - \frac{5}{2}x$$

$$5\begin{pmatrix} \frac{5}{2}x^2 - \frac{5}{2}x \\ 2 \end{pmatrix} = 5\frac{(\frac{5}{2}x^2 - \frac{5}{2}x)(\frac{5}{2}x^2 - \frac{5}{2}x)}{2} = 5(\frac{25}{4}x^4 - \frac{50}{4}x^3 + \frac{15}{4}x^2 - \frac{5}{2}x) = \frac{125}{4}x^4 - \frac{250}{4}x^3 + \frac{75}{4}x^2 - 2\frac{5}{2}x$$

This equation does not account for the biases used in construction to limit the application of constructive operators. This equation can be viewed more generally in terms of the levels of construction, *L*; the number of constructive operators, *Ops*; and the number of initial hypotheses, *X* as:

$$O(Ops^{2^L-1}x^{2^L})$$

This order of growth of the number of constructed hypotheses is valid for x ≥ 4. By way of example, Figure 25 shows how the number of hypotheses in the system increases with each level of construction for a system like HEC that uses 16 attributes in a feature.

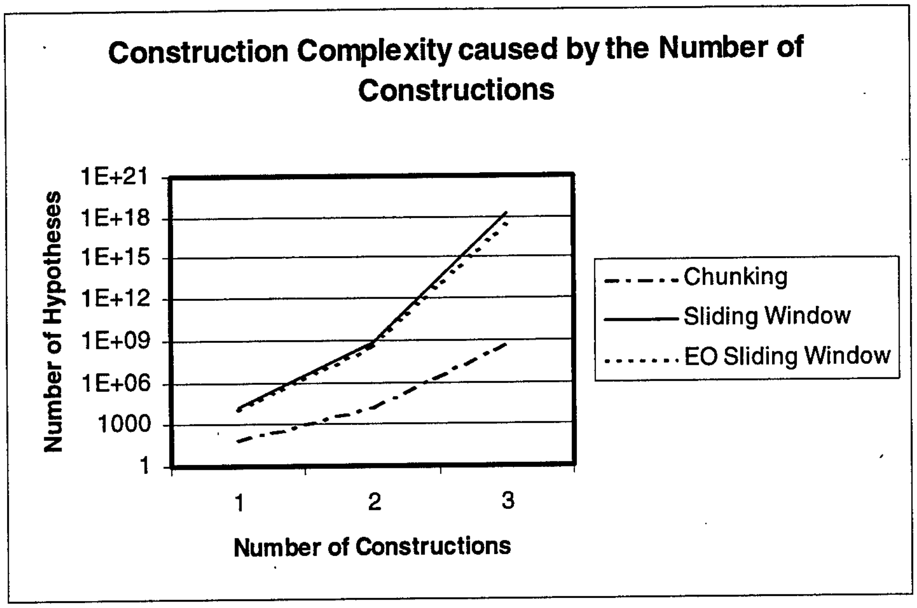

Figure 25 -- Complexity caused by Number of Constructions

5.3.3.2. Nondominated Set

A variety of techniques could be used to reduce the number of hypotheses in the system. More heuristics could be used to limit the number of hypotheses that are constructed. The label of a hypothesis with a high power and low purity could be switched from one concept to another. These hypotheses have an ability to classify many examples, however incorrectly; switching the label would dramatically improve the purity. Such a step is not needed since the corresponding hypothesis that has high power and high purity exists elsewhere in the hypothesis list. Hypotheses could be randomly chosen from the hypothesis list and used for construction. Such a stochastic process would have the same average performance of not ordering and would not guarantee any improvements.

HEC's ordering mechanism is based upon the nondominated set of hypotheses. This set is found by placing the hypotheses into "bins", a two-dimensional array that equally distributes the hypotheses based upon their power and purity score. A hypothesis is assigned to a bin based upon a mapping that translates the power and purity scores from the range [0.0 .. 1.0] into five bins with containing the ranges ([0.0 .. 0.20),(0.21 .. 0.40),(0.41 .. 0.60),(0.61 .. 0.80),(0.81..1.0]). Five bins were chosen to provide a coarse ordering, since a lack of existing empirical evidence impedes finer tuning. The system was designed to use a varying number of bins. With this system of bins, a hypothesis with a power score of 0.9 and a purity score of 1.0 would have the index (5,5) into the two dimensional array of bins. A hypothesis with a power score of 0.5 and a purity score of 1.0 would be placed into bin (3,5). The index into the bins is expressed in terms of the power index followed by the purity index, or (power, purity). The bin (5,5) is considered the "optimal" bin in this system since this is the location of the hypotheses with the highest power and purity scores. Figure 26 illustrates this system of bins and the location of the optimal bin.

The nondominated set is the set of hypotheses with scores that are greater than the hypotheses in the dominated set. In terms of the bins, the dominated set is the set of non-empty bins where another non-empty bin has a higher power index and a higher purity index. The one exception to this rule is that all filled bins in the column for the highest purity scores are included in the nondominated set. Since HEC is designed to learn classifiers for a concept, and purity measures the ability of the hypothesis to classify correctly, hypotheses with a high purity score should remain in the system.

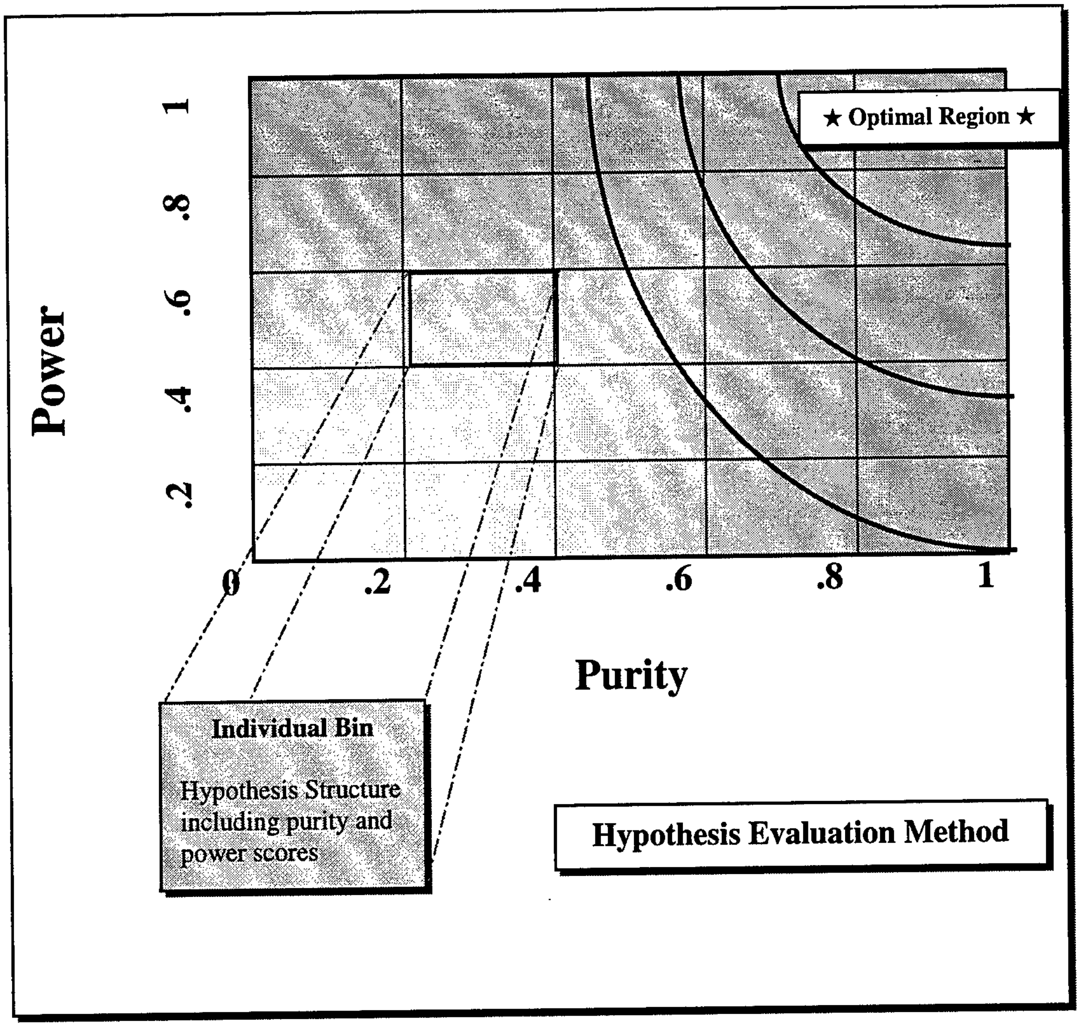
Power
1
.8
.6
.4
.2
0
.2
.4
.6
.8
1
Purity
★ Optimal Region ★
Individual Bin
Hypothesis Structure including purity and power scores
Hypothesis Evaluation Method

Figure 26 -- Hypothesis Evaluation Method Using Bins

5.3.3.3. Process / Algorithm

```
Index = indexes of previous ordering
HypoList = []
B = bins with evaluated hypotheses
PrevPur <- 0
For Pow = (5 .. 1)
        If PrevPur ≥ Index(Pow)
                PrevPur <- 0
        End If
        OldPur <- Index(Pow)
        For Pur <- (5 .. 1)
                If (Pur ≤ PrevPur) and (Pur < 5)
                        Exit loop
                End If
                If B(Pow,Pur).Count > 0
                        Append(HypoList, B(Pow,Pur).BinHypoList)
                        Index(Pow) <- Pur
                        If Pur < OldPur
                                PrevPur <- Pur
                                Exit Loop
                        End If
                End If
        End Loop
End Loop
Index = Specifies the bins that compromise the nondominated set
HypoList = Hypotheses in the nondominated set
```

Figure 27 -- Algorithm for Hypothesis Ordering

Figure 27 shows the algorithm used to order hypothesis stored in bins. The output of this algorithm is a list of hypotheses from the nondominated set. Additionally, an index is created that defines the location of the nondominated set in the bins. This index can be used if the nondominated set of hypotheses is not sufficient to construct classifiers of the examples. The index can be used to increase the membership of the nondominated set, increasing the number of hypotheses available for subsequent inductive steps.

5.3.3.4. Computational Complexity

Searching through the bins for the members of the nondominated set is accomplished through a doubly nested loop. Each loop iterates one time for every power row, or 5 times. The append operation can be accomplished in constant time through pointer manipulation. The overall complexity of the ordering step can be generalized to $O(b_{pow}b_{pur})$ where each *b* is the number of bins into which power and purity are divided.

5.3.4. Hypothesis Incorporation

Hypothesis incorporation is responsible for converting the hypotheses that correctly classify the examples into a form for acceptance into the knowledge base. The current version of MERCURY dose not fully implement this process. Future iterations of MERCURY should investigated algorithms for reducing the number of hypotheses needed to form detectors. Adding this incorporation capability will close the inductive loop.

Several fields of a hypothesis would be needed by the virus scanner, including *label, generation method,* and *features*. The hypothesis *score* could also be incorporated into the knowledge base, to aid the virus scanner in determining the relative worth of a classification.

When hypotheses are incorporated, they are sorted so that hypotheses that detect nonself are at the beginning of the list. This ordering ensures nonself byte patterns of files are detected before self byte patterns, improving the classification process. Following this sorting operation, the relative fields of a hypothesis are written to a text file, with one field written per line.

5.4. Knowledge Base Interface

The knowledge base is responsible for storing information about the detectors used to classify files as self or nonself. This knowledge base is currently structured as a flat text file consisting of the detectors for self and nonself. The nonself detectors are included first to ensure that viruses are detected with few misclassifications as self. This knowledge base can accept virus signatures from other files than HEC. Current virus signatures are comparable to nonself hypotheses selected through the *sliding window* selection rule. This ability ensures that existing virus detection knowledge is not lost using a system like MERCURY.

The only components of MERCURY that can access the knowledge base are HEC and the scanner. Future iterations of MERCURY will include interfaces for introducing new detectors through vaccination. Controlled user manipulation of the knowledge base is also an area of possible development, as well as virus updates received through the Computer Health System.

5.5. Scanner

The third and final component of MERCURY is the virus scanner. This scanner is responsible for determining the classification of a file based upon the self and nonself detectors created by HEC. The following sections discuss the use of self and nonself detectors to classify files on the system. In the current version of MERCURY, the scanner is not fully implemented. Reading the detectors from the knowledge base and determining the files to scan were not incorporated, while determining the classification of a file was. Several of the components of HEC can be reused in the scanner.

5.5.1. Reading Detectors

The detectors created by HEC are stored in the knowledge base. As discussed above, these detectors are stored in a flat file. Each detector consists of a label, feature, and generation method. The scanner reads this information from the file and creates a list of hypotheses based upon the hypothesis data structure from HEC.

5.5.2. Determining Files to Scan

Only certain files are susceptible to infection by viruses. This research is concerned with detecting file infector viruses in executable files, as compared to viruses that affect data files, for example macro viruses in the Microsoft Office product line.

Executable files are distinguished by the following file extensions: *exe, com, sys, dll, bat, vxd, cab* and *drv.* The list of files to scan is generated by searching the directory structure of the system for files with these extensions. This list is converted into the example list data structure used in HEC. In order to provide a basis for comparison against the list of detectors, each file is given the label of self.

5.5.3. Determining Classification of File

A file is classified by determining if any of the detectors can be found within it. This process is carried out by using the *test_example* function of HEC's evaluator. This function operates by comparing the list of examples to the list of hypotheses; in the scanner, the files to scan are compared to the list of detectors. Each file is opened and read, 16 bytes at a time, using the various selection methods. These bytes are compared to the features from the detectors. A flag is raised if there is a match based upon the detection scenarios below. The entire generation method and features of the detector must match for the file to be considered classified.

5.5.3.1. Detecting Unknown Files

A file is considered unclassified if no detector was able to classify it. In the fully implemented version of MERCURY, this file would be annotated as unclassified and sent to the virus expert to determine if the file is infected. Once the expert classified the file, HEC would learn a new detector for it.

5.5.3.2. Detecting Self Files

A file is classified as self if one or more self detectors are found in the file, and no nonself detectors are found. As discussed in Chapter Two, when a virus infects a file, the virus normally leaves large portions of the file intact. If the existence of any self detectors, without regard to the existence of nonself detectors, were used as the decision criteria, the system would be duped by "normal" viruses.

5.5.3.3. Detecting Nonself Files

A file is classified as nonself if *any* nonself detector is found. Once a file is detected as nonself, the user is informed of a possible virus. Future versions of the scanner will allow the user to remove the infection from the file, delete the file, or exclude the file from future scanning. The system could send the file to a virus expert in the Computer Health System, or if it was misclassified, send the file to HEC to learn a self detector for this file.

5.5.3.4. Detecting Previously Unseen Infectors

Previously unseen infectors are detected through the failure to recognize the self detectors and the predictive capability of the nonself detectors. When a virus infects a file, it moves portions of the file to different locations. Such an action can violate the conditions of the self detector. This violation may be caused by breaking the sequence of bytes used as a feature or breaking the relationships between the features established by the constructive operators. An infection may also be detected by a nonself detector that has the predictive capability to recognize features of similar viruses.

It is possible for viruses to go undetected. If a self detector is found and no nonself detectors are found, MERCURY could classify an infected file as self. The use of self and nonself detectors should improve the ability of MERCURY to combat these invaders by providing an extra layer of detectors the virus would need to elude. Future iterations of MERCURY should employ heuristics from current antivirus programs to supplement MERCURY's ability to detect previously unseen invaders.

5.6. Development Process

MERCURY was developed using a risk driven software process model called the spiral model. This process iterates through objectives, constraints, alternatives, risks, risk resolution, planning and commitment [Boe88]. Each step of the process is preceded by a risk analysis to determine if continuing the process will result in positive gains. The development of HEC followed this process.

The first iteration of the development process explored the feasibility of the algorithms used for construction. This iteration did not look at accessing files, but rather developing and integrating the processes of selection, construction, and evaluation. Through the use of Prolog, the algorithms were developed with respect to logic and purpose, rather than implementation. Construction used bitwise binary operators that manipulated the bits of the features through the AND, XOR, OR, and NOT operators. This method of construction was deemed ineffective, since it could misclassify files based upon the variety of byte combinations that could result in the same feature. Development in Prolog was halted due to execution speed considerations and the memory requirements of storing a large number of hypotheses. This iteration illustrates an alternative method to construction, representation of hypotheses and evaluation. The hypotheses were evaluated by converting the examples into a list of all possible derivations of an example using selection and construction.

The next iteration of the spiral investigated different alternatives to the induction algorithms with the intention of improving efficiency and classification. The current constructive operators replaced the bitwise operators. The bitwise operators were deemed ineffective based upon the lack of specificity in comparison to the original features. Finally, the program was translated to Ada, in order to take advantage of the structured nature of the algorithms and improved file access ability provided by this language. Another iteration resulted in more efficient evaluation routines, introduction of ordering, increased information hiding, lower coupling and greater cohesion.

5.7. Summary

This chapter highlighted the *prototyped* implementation of the *proposed* design of MERCURY. The chapter provided a detailed description of the components and processes within the *prototyped* version of MERCURY, including the *prototyped* versions of its three main components: the constructive engine, the scanner, and the knowledge base. The constructive induction, HEC, was decomposed into its four main processes: hypothesis generation, evaluation, ordering and incorporation. Biases that were incorporated into the learning process and the reasoning for their inclusion were investigated. These biases provide tidbits of background knowledge that aid the learning mechanism in determining the "best" classifier. Issues involving the knowledge base, and its integration into future iterations of MERCURY were addressed. The methodology used to scan for viruses was discussed, as well as possible solutions to detection challenges future iterations of MERCURY will encounter. Chapter Six presents the results of the tests run utilizing MERCURY. Chapter Seven draws conclusions from these analyses, and provides areas for system and methodological optimization, and future areas of research in these fields of study.

6. Analysis and Results

6.1. Introduction

Chapter Five provided a detailed description of the components and processes within the current version of MERCURY, focusing attention on the constructive induction engine named HEC. This chapter presents the results of running various testing scenarios utilizing MERCURY, in order to determine the effectiveness of the constructive induction approach applied to virus detection.

First, the eleven test cases are explained in terms of their creation and testing purpose. The performance of MERCURY is analyzed in five dimensions: time, space, power and purity, coverage and process optimization. Each of these dimensions are described in their importance for rejecting or supporting the hypotheses stated in Chapter One. The primary research hypothesis as related to MERCURY is that constructive induction provides a suitable learning mechanism for the virus detector system of an individual computer system. In support of this primary hypothesis, the virus feature hypothesis conjectured byte patterns can be used as the basis of a constructive induction based computer virus detector. Additionally, the constructive operator hypothesis stated logical and spatial operators can be used for constructing new attributes for the computer virus detector. Conclusions from these analyses are presented in Chapter Seven along with future areas of research.

6.2. Test Cases

Testing of MERCURY was accomplished through analyzing the hypotheses generated for all eleven test cases. These test cases were divided into two groupings: laboratory and operational. The laboratory test cases were designed to test MERCURY's functionality under certain expected situations along controlled dimensions. The operational test cases utilized segments of actual application programs to test MERCURY's functionality in simulated "real world" situations.

Each test case was composed of eight files labeled self and two files labeled nonself. This composition was chosen to reflect the small number of files that may be infected on a computer. Each file was 100 bytes in length, based upon the large time and space growth needed for generating, evaluating and ordering hypotheses.

The laboratory test cases were designed to investigate MERCURY's performance in particular situations. The files that composed each test case do not reflect files that are used in a computer; rather, these files reflect situations that might be encountered or particular machine learning problems. Since these test cases were used to validate and verify MERCURY, the analyses presented in the subsequent sections will focus on the operational test cases. Table 18 provides an overview of the laboratory test cases.

Table 18 -- Test Cases 1 - 8

Number	Structure	Purpose
1	*Self* - All 1's *Nonself* - All 0's	Does detection work?
2	*Self* - All 1's *Nonself* - Random characters	Does MERCURY detect repeated patterns?
3	*Self* - Random characters without *y* *Nonself* - Random characters with *y*	Does MERCURY induce classifier for infrequent patterns?
4	*Self* - Have equal number of 1's and 0's *Nonself* - Random characters with unequal number of 1's and 0's	Does MERCURY detect parity of bytes?
5	*Self* - Contain pattern *y* same distance apart *Nonself* - Contain pattern *y* varying distance apart	Does MERCURY detect spatially and logically?
6	*Self* - same as *Nonself*	Does MERCURY's evaluation process work?
7	*Self* and *nonself* are complement of each other	Can MERCURY induce detector for absolute position?
8	*Self* - Randomly generated string *Nonself* - Randomly generated string	Can MERCURY detect patterns in random strings?

The operational test cases investigated MERCURY's ability to induce detectors for segments of application programs. The files in this test case were created by extracting a 100 byte segment approximately 2,000 bytes offset from the beginning of the file. This segment of bytes was chosen to increase the probability of detection based upon patterns in the binary application code, rather than patterns in binary libraries included in the beginning of many applications. Table 19 shows the structure and purpose of each operational test case.

Each test run collected information on: the time needed for selection and construction; the coverage of the examples by the hypotheses; and the *label*, *generation method*, and *score* fields of the hypotheses.

Table 19 -- Test Cases 9 - 11

Number	Structure	Purpose
9	*Self* - Randomly chosen programs *Nonself* - Randomly chosen programs	Does MERCURY detect patterns in programs?
10	*Self* - Programs copyrighted by companies other than Microsoft *Nonself* - Programs copyrighted by Microsoft	Does MERCURY detect patterns in programs from different companies?
11	*Self* - Randomly chosen programs *Nonself* - File infector viruses	Does MERCURY detect viral patterns?

6.3. Time

The first dimension of MERCURY analyzed was the time the constructive induction process utilized to generate, evaluate and order the hypotheses. Two time parameters were considered important for each test case: the time needed for selective induction and the time needed for constructive induction. The following subsections analyze the time results for the laboratory and operational test cases. These results were obtained by running each test on one type of computer and cross validating some of these results on another type of computer. The first type of computer was an Intel Pentium 200MHz computer with 64MB of RAM running Windows 95. The second type of computer was an Intel Pentium II 350MHz with 64MB of RAM running Windows 95.

The laboratory test cases illustrated the difference in time requirements for selective and constructive induction. Selective induction was run on the first hardware platform, and took an average of 1.5 minutes to complete while constructive induction took an average of 1338.5 minutes to complete. These results were not cross validated based upon the small expectations of occurrence in operation. The graphical depiction in Figure 28, shows the disparity in time needed for selection and construction.

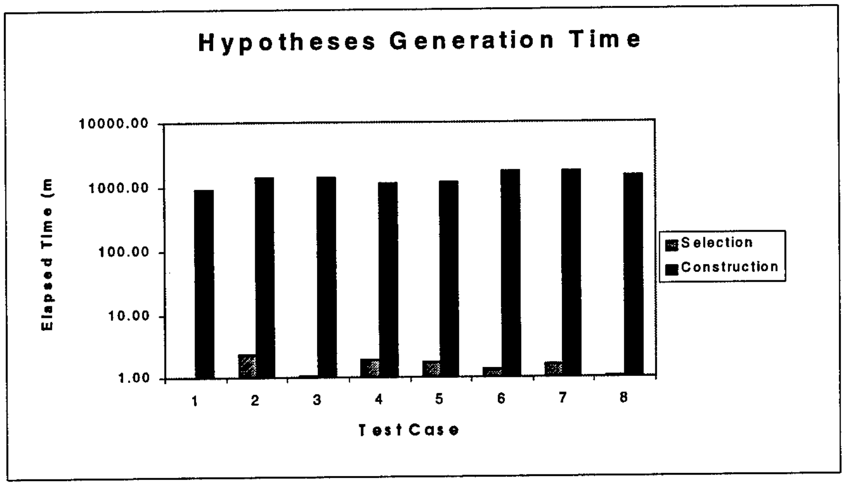

Figure 28 -- Hypotheses Generation Time for Test Cases 1-8

Similar to the laboratory test cases, there was a significant difference in the induction times between selection and construction for the operational test cases. Each time result was cross validated to investigate the effect of hardware upon the performance of MERCURY. Selective induction for these test cases took 5.4 and 2.8 minutes for each testing platform respectively. Constructive induction took 4797.5 and 3531.9 minutes respectively for each hardware platform. These results are graphically depicted in Figure 29.

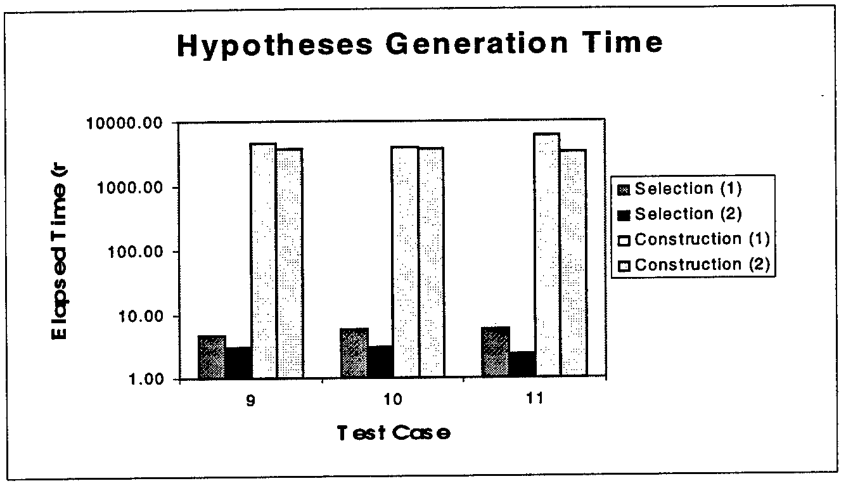

Figure 29 -- Hypotheses Generation Time for Test Cases 9-11

6.4. Space

The next performance dimension of MERCURY is the number of hypotheses that were induced. The results of this section were based upon a frequency analysis of the hypotheses that remained following hypothesis generation, evaluation and ordering.

Several aspects were important to the space performance of MERCURY. The first consideration was the composition of hypotheses generated during selection. The next consideration was the composition of hypotheses generated during construction by each method. The third consideration was the composition of hypotheses generated during construction by each operator. The final consideration was the total number of hypotheses created by selective and constructive induction, combined.

The first aspect of space performance is the composition of hypotheses generated by each selection rule in selective induction. Across all the laboratory test cases, 4% of all hypotheses were generated by the chunking selection rule, 44% by the every other byte sliding window selection rule, and 52% by the sliding window rule. Across all the operational test cases, 4% of all hypotheses were generated by the chunking selection rule, 43% by the every other byte sliding window selection rule, and 53% by the sliding window rule. These percentages agreed with the space predictions in Chapter Five. Additionally, these percentages were similar to the results of the laboratory test cases. This similarity was a result of the algorithm used to generate hypotheses, rather than the effect of data. These percentages agreed with the space predictions in Chapter Five.

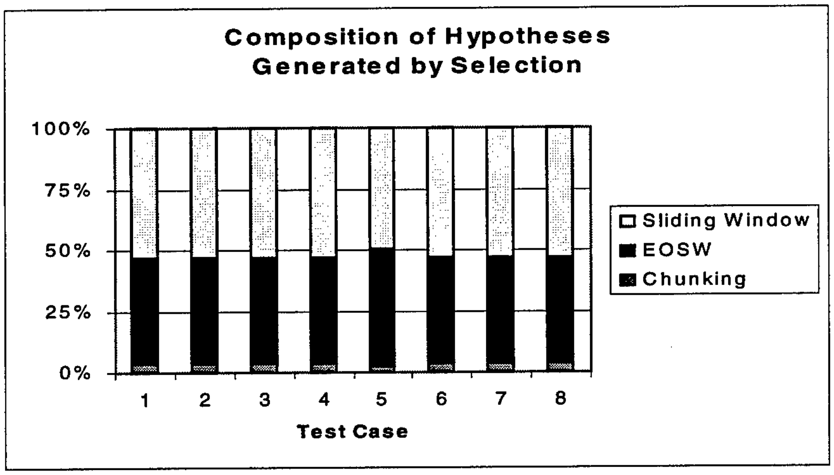

Figure 30 -- Composition of Hypotheses Generated by Selection for Test Cases 1-8

The second important aspect of space performance is the composition of constructed hypotheses. This aspect was useful for showing a change in the composition of the hypothesis list when the selected hypotheses were used for construction. Across all laboratory cases, 0.2% of all constructed hypotheses were based upon chunking selection rule hypotheses, 40% were based upon every other byte sliding window selection rule hypotheses, and 59% were based upon sliding window selection rule hypotheses. Across all operational cases, 0.3% of all constructed hypotheses were based upon chunking selection rule hypotheses, 39% were based upon every other byte sliding window selection rule hypotheses, and 60% were based upon sliding window selection rule hypotheses. The results between the laboratory and operational tests were essentially equivalent. Again, these results showed the construction algorithm as the determinant for the composition of hypotheses, rather than the data that was used. The results for the laboratory test cases are depicted in Figure 31.

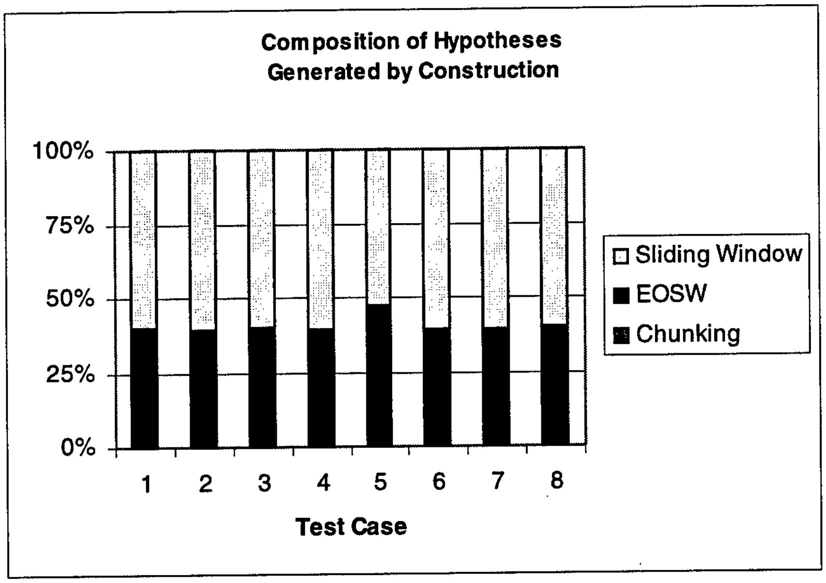

Figure 31 -- Composition of Hypotheses Generated by Construction for Test Cases 1-8

The third space performance aspect is the composition of hypotheses constructed by the constructive operators. Figures 32 and 33 show this composition. These results showed that the OR and XOR constructive operators constructed a majority of hypotheses that remained in the system after hypotheses ordering. The results for these operators reflected the effect of fewer biases that constricted the choice of hypotheses used for construction.

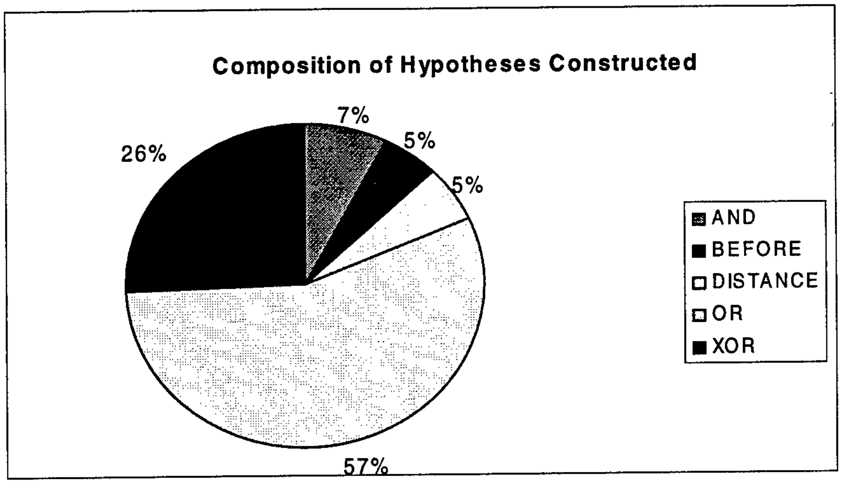

Figure 32 -- Operator Composition of Constructed Hypotheses for Test Cases 1 – 8

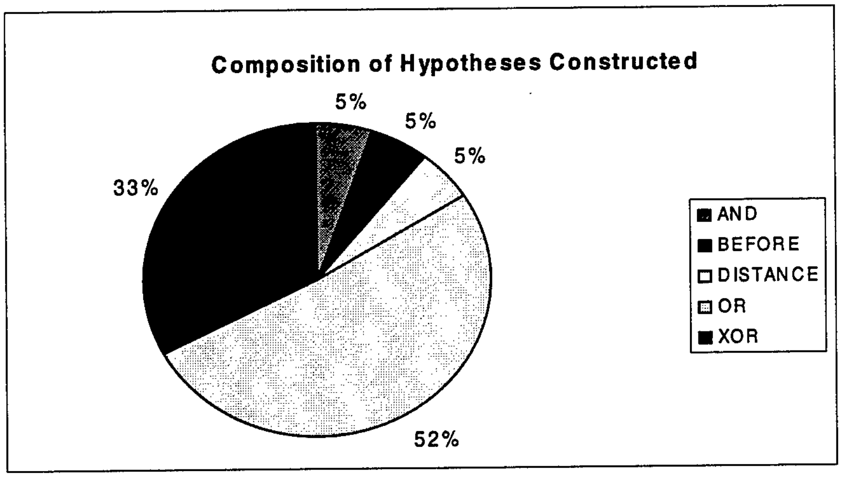

Figure 33 -- Operator Composition of Constructed Hypotheses for Test Cases 9 - 11

The final aspect of the space performance of MERCURY is the number of hypotheses generated by selection and construction, combined. Based upon the combinatorial process of constructing hypotheses, the number of constructed hypotheses was much greater than the number of selected hypotheses. The total number of hypotheses was dependent upon the examples. The range of selected hypotheses was

750-1,600 hypotheses, while the range of constructed hypotheses was 250,000 to 910,000 hypotheses as shown in Figures 34 and 35.

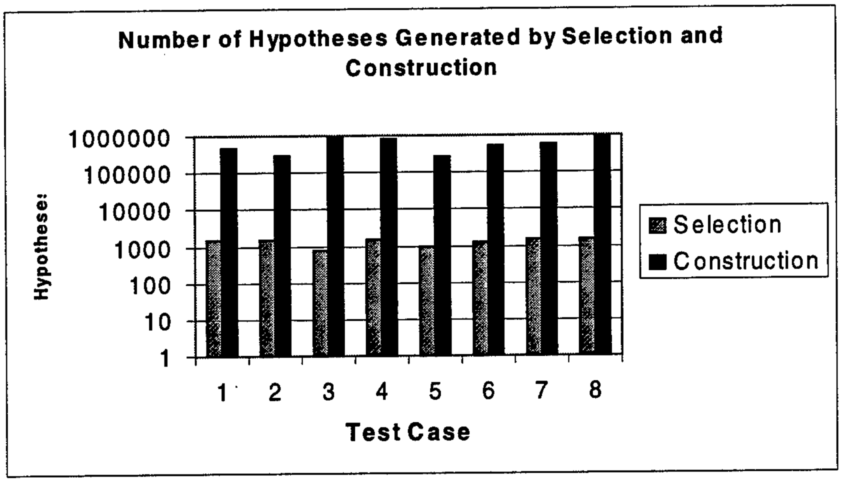

Figure 34 -- Number of Hypotheses Generated for Test Cases 1 – 8

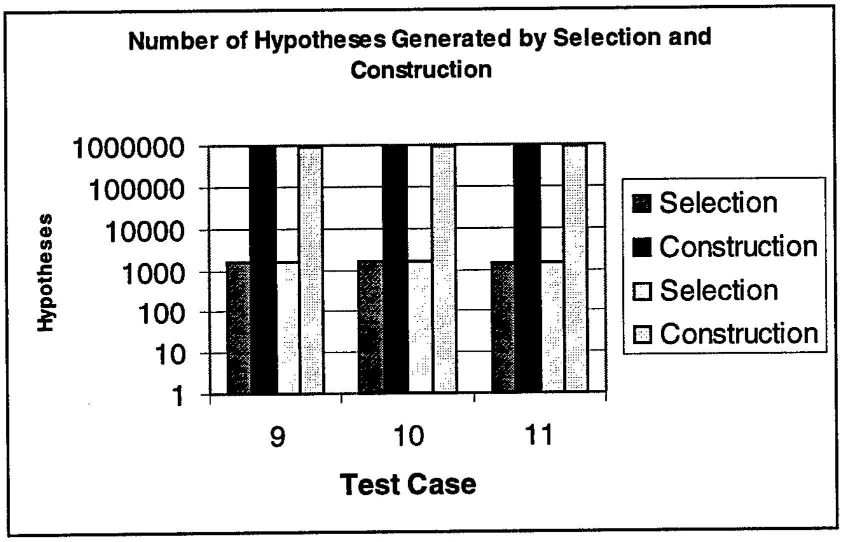

Figure 35 -- Number of Hypotheses Generated for Test Cases 9 - 11

6.5. Power and Purity

The next performance dimension of MERCURY is power and purity. As discussed in Chapter Five, power measured the number of examples that the hypothesis classified, either correctly or incorrectly, while purity gauged the correctness of the classifications that the hypothesis made. These measurements provided a local view of the performance of MERCURY. The following subsections explore the power and purity of hypotheses in MERCURY from the laboratory and operational test cases. This exploration looks at five aspects of power and purity: selection, construction, analysis of selection methods, analysis of operators, and overall results. This section provides a foundation for the process optimization section, which will explore the effects of the different system parameters upon power and purity.

The power and purity scores were the weighted average of the raw power and purity scores that were stored in the hypotheses, in relation to the number of hypotheses with that score combination. These hypotheses were considered in the same hypothesis class. This weighted average was used to account not only for the best performers, but also for the amount of processing the system required to induce those hypotheses.

The power and purity scores were analyzed by self and nonself. This distinction was necessary due to the composition of the test cases, which have eight self and two nonself files. Without this distinction, nonself hypotheses with the optimal power score of 20% and an optimal purity score of 100% would not be distinguished from suboptimal self hypotheses with the same scores. The power and purity scores of each hypothesis were weighted by the number of examples in the concept that the hypothesis should be able to classify, allowing for direct comparison of the values. A consequence of this weighting was that overclassification is indicated by a power score greater than 100%. MERCURY should be able to induce hypotheses with a power score greater than 12.5% for self and 50% for nonself. These scores indicate hypotheses that only detect one example, reflecting a performance equivalent to chance.

6.5.1. Laboratory Test Cases

The first aspect of power and purity is the average power and purity scores for selection. Based upon the laboratory test cases, the average power score for self was 66.5% and 86.9% for nonself, while the average purity score for self was 96.0% and 100.0% for nonself. These values indicated that a large number of the selected hypotheses were good detectors. This inflated result was possibly due to the contrived nature of the laboratory test cases. The average scores for the different selection rules are illustrated in Figure 36 and 37. These graphs show that the selection rules appeared to have no relation to the power, while the every other byte sliding window selection rule appeared to be a poor performer for purity.

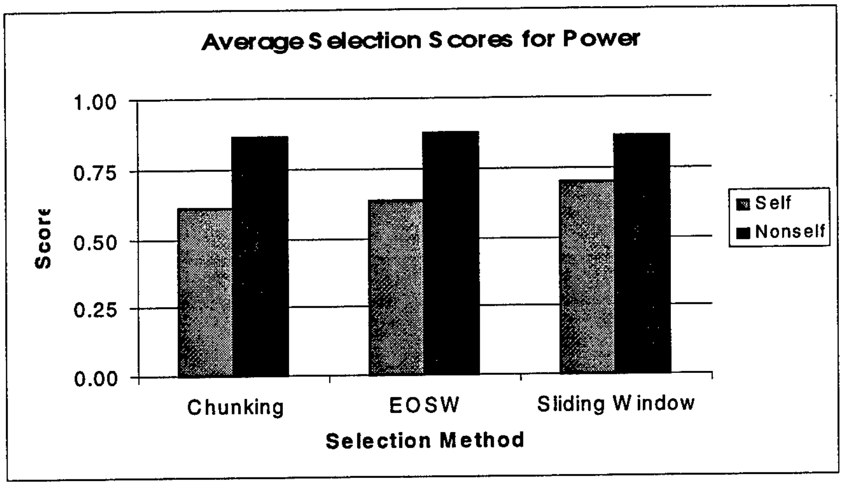

Figure 36 -- Average Selection Scores for Power for Test Cases 1 – 8

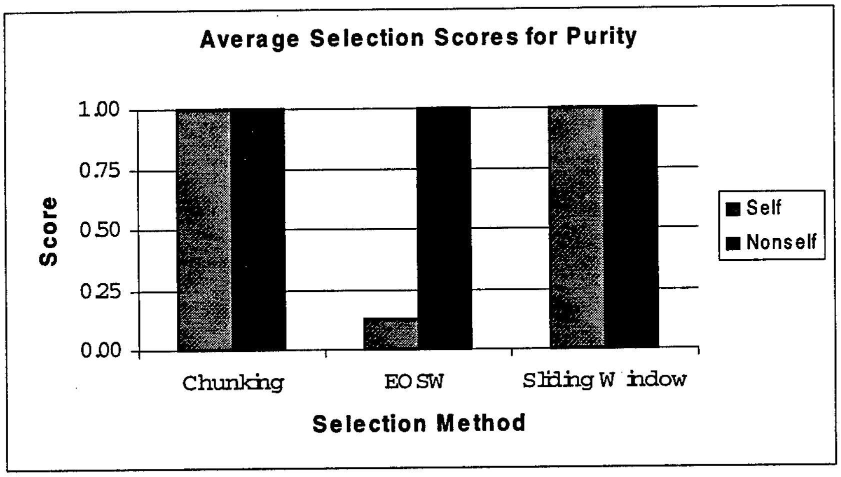

Figure 37 -- Average Selection Scores for Purity for Test Cases 1 - 8

While the average power and purity scores show the overall state of the system, the detectors utilized as signatures will most likely be the hypotheses with the maximal power and purity scores. The maximal power score was 100% for both self and nonself, while the maximal purity score was 97% for self and 100% for nonself. These results were the maximums across all the test cases.

The second aspect of power and purity is the average power and purity scores for construction. Based upon the laboratory test cases, the average power score for self was

60.8% and 149.6% for nonself, while the average purity score for self was 96.9% and 98.3% for nonself. These values decreased in relation to their counterparts for selection. Test cases 5 and 6 were designed to reduce the performance of the constructive operators. Test case 5 was designed to reduce the effectiveness of the logical operators, while increasing the effectiveness of the spatial operators. Since the space performance analysis showed that more hypotheses constructed with logical operators were kept in the system, this test case artificially inflated overclassification. Test case 6 was designed to eliminate the number of possible unique patterns between self and nonself. With no unique patterns to distinguish, this test case forced all hypotheses to overclassify. Statistical evidence of these conclusions is presented in the process optimization section of this chapter.

The average scores for the different selection rules are illustrated in Figure 38 and 39. These graphs showed that constructed hypotheses, based upon the sliding window selection rule, had the most overclassification, with a nonself average power of 184%. Additionally, constructed hypotheses based upon the chunking selection rule and the every other byte sliding window selection rule have similar power and purity scores.

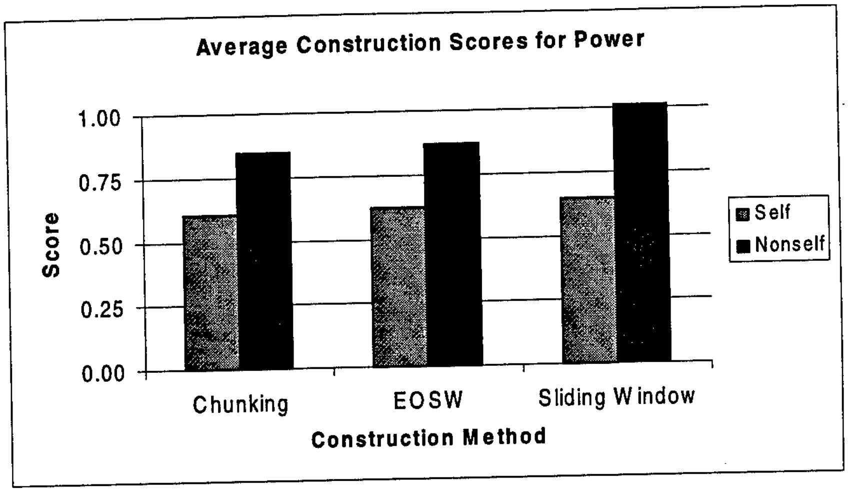

Figure 38 -- Average Construction Scores for Power for Test Case 1 – 8

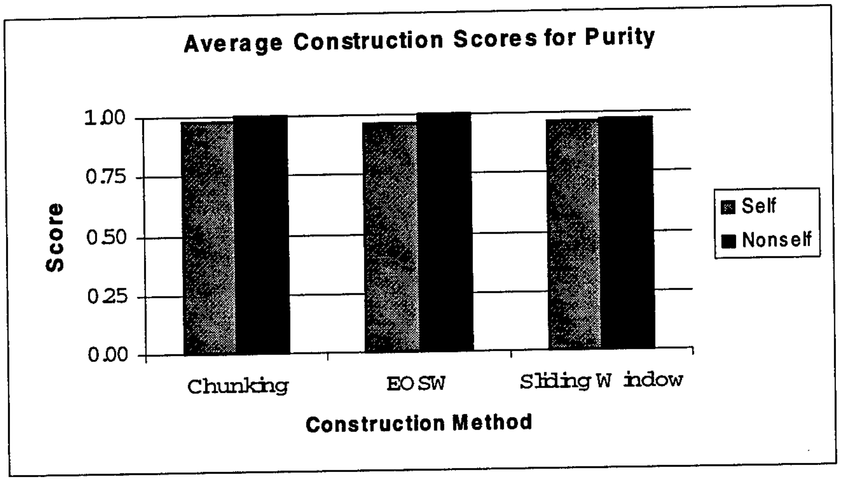

Figure 39 -- Average Construction Scores for Purity for Test Case 1 - 8

Another aspect of the power and purity performance of construction is the choice of constructive operators. The average performance of the constructive operators is shown in Figures 40 and 41. Figure 40 fails to show the full extent of the overclassification caused by the AND operator, which had an average power of 273%.

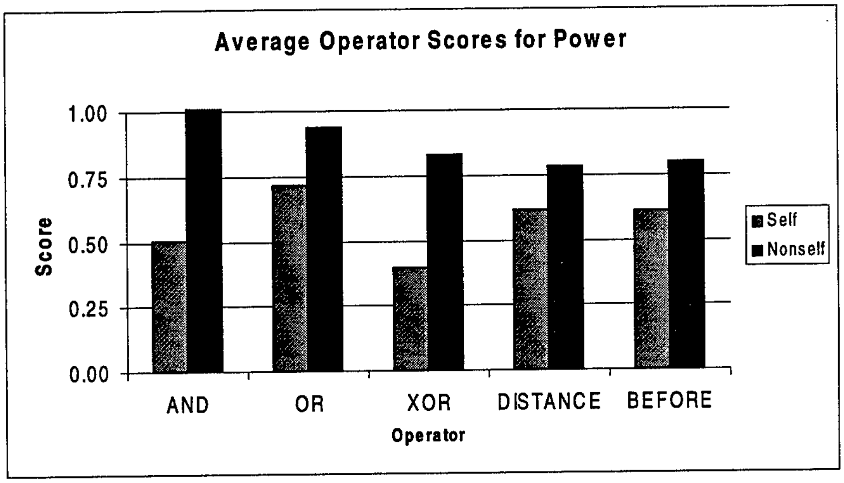

Figure 40 -- Average Operator Scores for Power for Test Cases 1 – 8

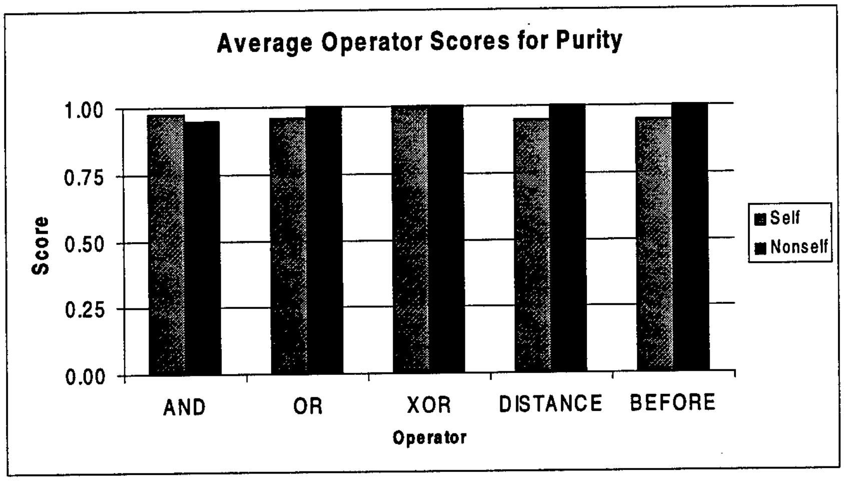

Figure 41 -- Average Operator Scores for Purity for Test Cases 1 - 8

		Average Power		Average Purity		Max Power		Max Purity		Min Power		Min Purity	
		Self	Nonself	Self	Nonself	Self	Nonself	Self	Nonself	Self	Nonself	Self	Nonself
Selection	Chunking	61.0%	86.3%	100.0%	100.0%	100.0%	100.0%	97.0%	100.0%	12.5%	50.0%	80.0%	100.0%
	EOSW	63.5%	87.6%	12.5%	100.0%	100.0%	100.0%	80.0%	100.0%	12.5%	50.0%	96.0%	100.0%
	Sliding Window	69.9%	86.3%	100.0%	100.0%	80.0%	100.0%	96.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	Selection Total	*66.5%*	*86.9%*	***96.0%***	***100.0%***	*100.0%*	*100.0%*	***97.0%***	***100.0%***	*12.5%*	*50.0%*	***80.0%***	***100.0%***
		70.6%											
Construction	Logical Chunking	60.1%	87.9%	97.5%	100.0%	125.0%	100.0%	100.0%	100.0%	12.5%	50.0%	80.0%	100.0%
	Spatial Chunking	61.2%	74.5%	95.1%	100.0%	125.0%	100.0%	100.0%	100.0%	12.5%	50.0%	80.0%	100.0%
	Chunking Total	***60.2%***	***84.3%***	***97.4%***	***100.0%***	***125.0%***	***100.0%***	***100.0%***	***100.0%***	***12.5%***	***50.0%***	***80.0%***	***100.0%***
	Logical EOSW	62.1%	90.1%	96.8%	100.0%	125.0%	100.0%	100.0%	100.0%	12.5%	50.0%	80.0%	100.0%
	Spatial EOSW	62.4%	79.6%	94.5%	100.0%	125.0%	100.0%	100.0%	100.0%	12.5%	50.0%	80.0%	100.0%
	EOSW Total	***62.2%***	***86.9%***	***96.6%***	***100.0%***	***125.0%***	***100.0%***	***100.0%***	***100.0%***	***12.5%***	***50.0%***	***80.0%***	***100.0%***
	Logical Sliding	59.8%	221.5%	97.3%	96.3%	125.0%	500.0%	100.0%	100.0%	12.5%	50.0%	80.0%	80.0%
	Spatial Sliding	60.5%	78.4%	94.9%	100.0%	125.0%	100.0%	100.0%	100.0%	12.5%	50.0%	80.0%	100.0%
	Sliding Total	***64.7%***	***184.0%***	***96.0%***	***97.3%***	***125.0%***	***500.0%***	***100.0%***	***100.0%***	***12.5%***	***50.0%***	***80.0%***	***80.0%***
	Construction Total	*60.8%*	*149.6%*	***96.9%***	***98.3%***	*125.0%*	*500.0%*	***100.0%***	***100.0%***	*12.5%*	*50.0%*	***80.0%***	***80.0%***
		78.6%											
Operators	AND	50.4%	273.4%	97.6%	94.8%	125.0%	500.0%	100.0%	100.0%	12.5%	50.0%	80.0%	80.0%
	OR	71.7%	93.7%	95.8%	100.0%	125.0%	100.0%	100.0%	100.0%	12.5%	50.0%	80.0%	100.0%
	XOR	39.4%	82.9%	99.8%	100.0%	112.5%	100.0%	100.0%	100.0%	12.5%	50.0%	80.0%	100.0%
	Logical Operator Total	***60.7%***	***176.7%***	***97.1%***	***97.6%***	***125.0%***	***500.0%***	***100.0%***	***100.0%***	***12.5%***	***50.0%***	***80.0%***	***80.0%***
	DISTANCE	61.3%	78.3%	94.7%	100.0%	125.0%	100.0%	100.0%	100.0%	12.5%	50.0%	80.0%	100.0%
	BEFORE	61.2%	79.7%	94.7%	100.0%	125.0%	100.0%	100.0%	100.0%	12.5%	50.0%	80.0%	100.0%
	Spatial Operator Total	***61.2%***	***79.0%***	***94.7%***	***100.0%***	***125.0%***	***100.0%***	***100.0%***	***100.0%***	***12.5%***	***50.0%***	***80.0%***	***100.0%***
	Operator Total	*60.8%*	*149.6%*	***96.9%***	***98.3%***	*125.0%*	*500.0%*	***100.0%***	***100.0%***	*12.5%*	*50.0%*	***80.0%***	***80.0%***
		78.6%											

Table 20 -- Summary of Results for Test Cases 1- 8

An overall analysis of the results for the laboratory test case is provided in Table 20. These results showed that the average power score increased from selection to construction, 70.6% to 78.6%, respectively. The average power score was based upon a proportion of the averages for self and nonself power and the number of self and nonself examples. This increase indicated that the constructive induction process appeared to help learn detectors for files. However, the average purity score decreased for both self and nonself examples. A comparison of maximum power for selection and construction provided little information since overclassification resulted in power scores greater than 100%. Looking at the maximum values when overclassification exists falsely inflated the worth of the overclassified parameter.

6.5.2. Operational Test Cases

The operational test cases reflected a "real world" view of the performance of MERCURY. The individual test cases were not designed to reduce the performance of one aspect of the system in order to test another aspect, as occurred with the laboratory test cases. This section will analyze these test cases with respect to the power and purity scores from selection, construction, construction with respect to each selection method, construction with respect to each operator. It also provides an overall view of the test cases.

The first aspect of the power and purity performance is the average power and purity for selection. All hypotheses induced for the operational test cases had a purity of 100% for both self and nonself. Since purity was not a varying parameter in the operational test cases, it will not be discussed further in this section.

During the operational test cases, MERCURY was able to generate detectors with an average power score of 12.7% for self and 50% for nonself. These power scores were equivalent to the chance scores of 12.5% for self and 50% for nonself. The selection rules did not vary the average power scores, as can be seen in Figure 42. The maximum power for self hypotheses was 37.5% and 50% for nonself detectors.

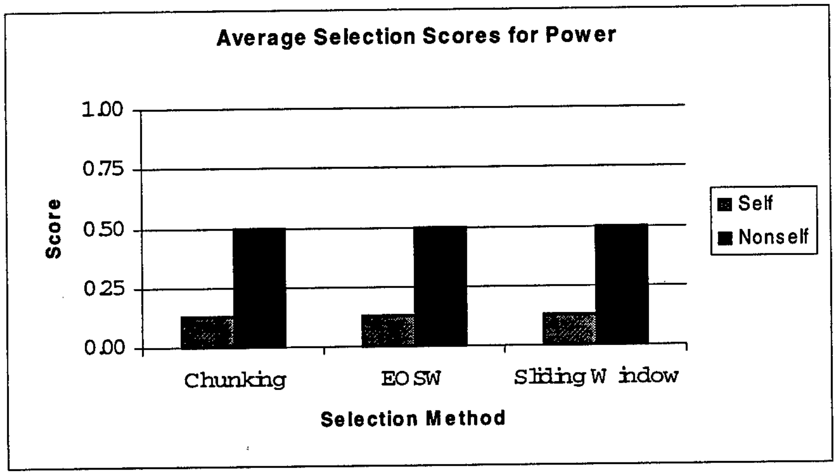

Figure 42 -- Average Selection Scores for Power for Test Cases 9 - 11

The next aspect of the power and purity performance of the operational test cases is the average power for construction, which were 22.6% for self and 66.9% for nonself. These scores were not equivalent to the chance scores of 12.5% for self and 50% for nonself. Additionally, the maximum power score was 62.5% for self and 100% for nonself. The average and maximum power scores both increased from selection to construction.

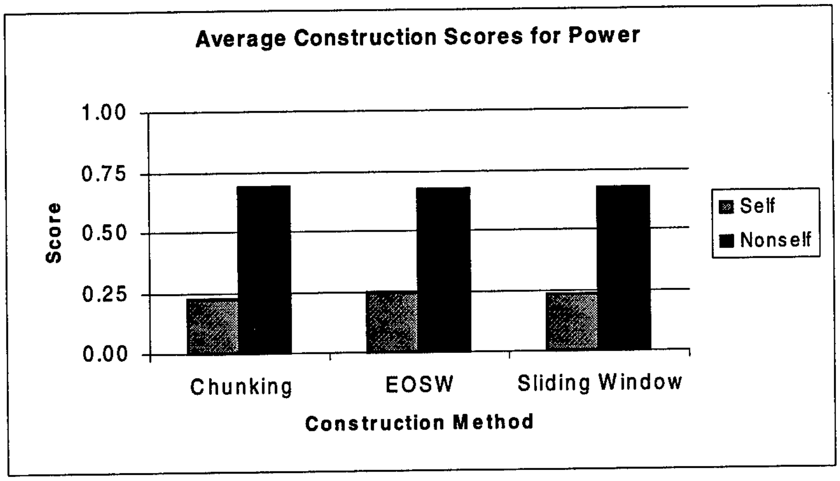

Figure 43 -- Average Construction Scores for Power for Test Cases 9 - 11

The above figure shows that the power scores for constructed hypotheses vary slightly with the selection rule, upon which the hypothesis is based. Table 21 shows that the best performing hypotheses were constructed using the sliding window selection rule; their maximum power was 62.5% for self and 100% for nonself. The process optimization section will continue to explore the relationship between power and the various selection rules.

The next aspect of the power performance of the operational test cases is the effect of the constructive operators. Figure 44 portrays the relationship between the constructive operators and power. The OR and XOR operators have different power scores than the other operators. OR had an average power score of 23.9% for self and 75.2% for nonself, while XOR had an average power score of 25.4% for self and 100% for nonself. The maximum power scores for both OR and XOR were 62.5% for self and 100% for nonself. These scores were also the maximum across all the operators.

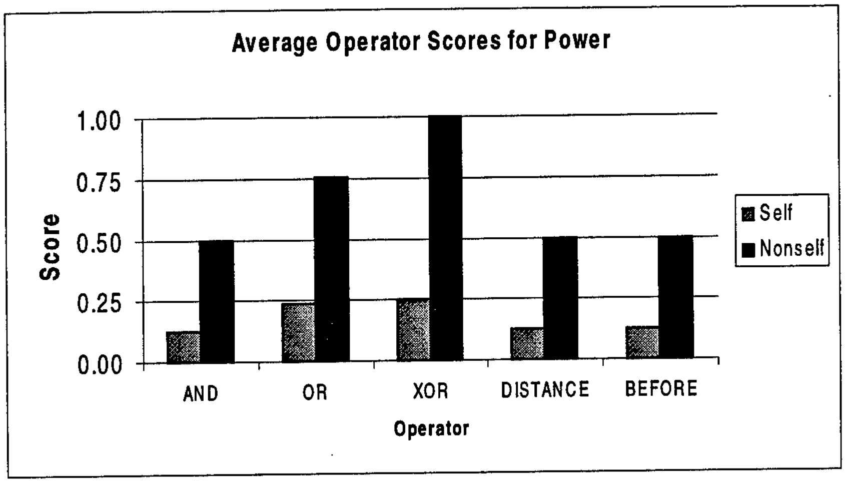

Figure 44 -- Average Operator Scores for Power for Test Cases 9 -11

The overall power and purity performance can be found in Table 21. As stated above, the purity of all the hypotheses was 100%. The average power across the self and nonself concepts increased from 20.2% for selection to 31.5% for construction. This increase shows that MERCURY was able to create better hypotheses. This finding is supported by the increase in maximum power for self from 37.5% to 62.5% and for nonself from 50% to 100%.

		Average Power		Average Purity		Max Power		Max Purity		Min Power		Min Purity	
		Self	Nonself	Self	Nonself	Self	Nonself	Self	Nonself	Self	Nonself	Self	Nonself
Selection	Chunking	12.5%	50.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	EOSW	12.6%	50.0%	100.0%	100.0%	25.0%	50.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	Sliding Window	12.9%	50.0%	100.0%	100.0%	37.5%	50.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	Selection Total	*12.7%*	*50.0%*	***100.0%***	***100.0%***	*37.5%*	*50.0%*	***100.0%***	***100.0%***	*12.5%*	*50.0%*	***100.0%***	***100.0%***
		20.2%											
Construction	Logical Chunking	23.7%	77.5%	100.0%	100.0%	25.0%	100.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	Spatial Chunking	12.5%	50.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	Chunking Total	***22.6%***	***69.0%***	***100.0%***	***100.0%***	***25.0%***	***100.0%***	***100.0%***	***100.0%***	***12.5%***	***50.0%***	***100.0%***	***100.0%***
	Logical EOSW	23.5%	75.2%	100.0%	100.0%	37.5%	75.2%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	Spatial EOSW	12.5%	50.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	EOSW Total	***24.9%***	***67.0%***	***100.0%***	***100.0%***	***37.5%***	***75.2%***	***100.0%***	***100.0%***	***12.5%***	***50.0%***	***100.0%***	***100.0%***
	Logical Sliding	24.1%	75.2%	100.0%	100.0%	62.5%	100.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	Spatial Sliding	12.5%	50.0%	100.0%	100.0%	37.5%	50.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	Sliding Total	***22.8%***	***66.8%***	***100.0%***	***100.0%***	***62.5%***	***100.0%***	***100.0%***	***100.0%***	***12.5%***	***50.0%***	***100.0%***	***100.0%***
	Construction Total	*22.6%*	*66.9%*	***100.0%***	***100.0%***	*62.5%*	*100.0%*	***100.0%***	***100.0%***	*12.5%*	*50.0%*	***100.0%***	***100.0%***
		31.5%											
Operators	AND	12.5%	50.0%	100.0%	100.0%	37.5%	50.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	OR	23.9%	75.2%	100.0%	100.0%	62.5%	100.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	XOR	25.4%	100.0%	100.0%	100.0%	62.5%	100.0%	100.0%	100.0%	12.5%	100.0%	100.0%	100.0%
	Logical Operator Total	***23.9%***	***75.2%***	***100.0%***	***100.0%***	***62.5%***	***100.0%***	***100.0%***	***100.0%***	***12.5%***	***50.0%***	***100.0%***	***100.0%***
	DISTANCE	12.5%	50.0%	100.0%	100.0%	37.5%	50.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	BEFORE	12.5%	50.0%	100.0%	100.0%	37.5%	50.0%	100.0%	100.0%	12.5%	50.0%	100.0%	100.0%
	Spatial Operator Total	***12.5%***	***50.0%***	***100.0%***	***100.0%***	***37.5%***	***50.0%***	***100.0%***	***100.0%***	***12.5%***	***50.0%***	***100.0%***	***100.0%***
	Operator Total	*22.6%*	*66.9%*	***100.0%***	***100.0%***	*62.5%*	*100.0%*	***100.0%***	***100.0%***	*12.5%*	*50.0%*	***100.0%***	***100.0%***
		31.5%											

Table 21 -- Summary of Results for Test Cases 9 - 11

6.6. Coverage

The fourth dimension of the performance of MERCURY is coverage. This measurement provided information about the global behavior of MERCURY. As defined in Chapter Five, coverage is the number of hypotheses that classify, misclassify and fail to classify each example. Coverage can be used to determine the classification rate, or the percent of examples that are classified, misclassified or not classified by the hypotheses. The following subsections provide a discussion of the classification rate of MERCURY.

The power and purity results for the laboratory test cases showed that some hypotheses overclassified. The coverage results in Figure 45 showed that for test cases 5 and 6 the nonself files were misclassified. In test case 5, hypotheses were induced that either classified or failed to classify the self examples. In addition, a large number of hypotheses were induced that misclassified the nonself files, with a smaller number either failing to classify or correctly classifying these files. These results indicated that hypotheses were induced having the ability to classify the nonself files, while not misclassifying the self files. While a definitive conclusion can not be reached without determining *which* hypothesis classified *which* examples, it is probable that a smaller subset of the hypothesis list from test case 5 could have classified each example in the training set. However, no subset could be derived for test case 6. With the files the same for both self and nonself, finding a detector with the ability to distinguish the two concepts is impossible. The results concurred with this analysis, since no hypotheses were able to classify the nonself examples correctly.

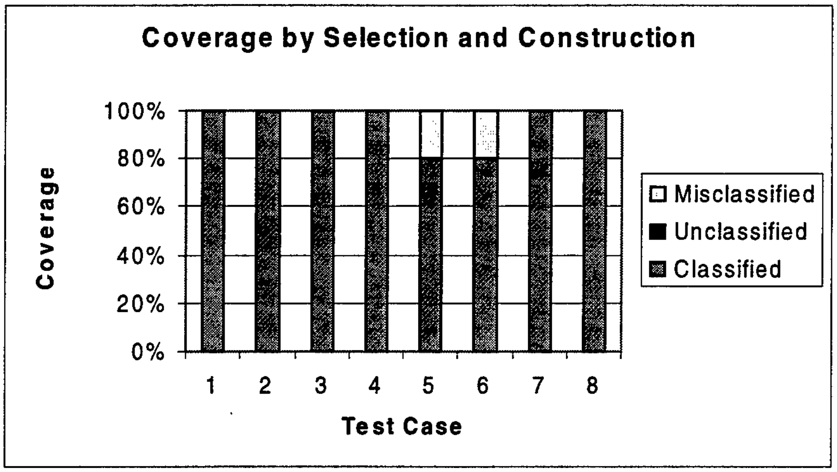

Figure 45 -- Coverage for Test Cases 1 - 8

The coverage results for the operational test cases are graphed with respect to classification in Figure 46. These results show that all hypotheses had 100% purity, each example was classified and there were no misclassifications.

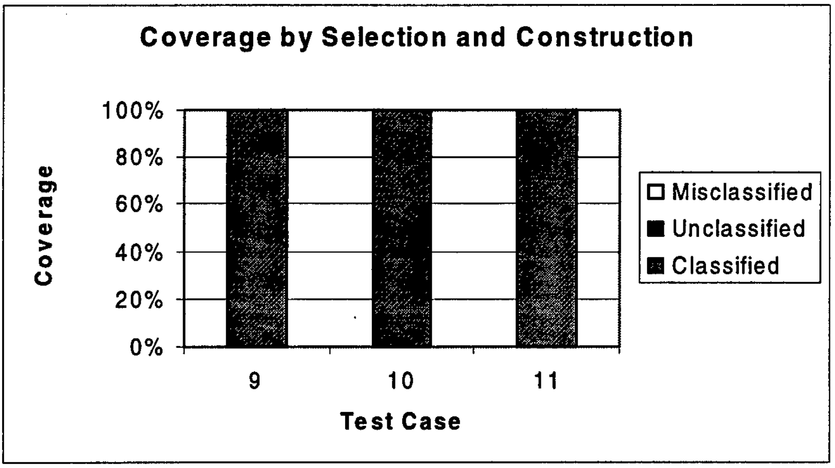

Figure 46 -- Coverage for Test Cases 9 –11

6.7. Process Optimization Through Response Surface Methodology Techniques

Response surface methodology (RSM) is a collection of statistical and mathematical techniques useful for *developing, improving, and optimizing processes* [MyM95]. This technique is useful in applications where several input variables potentially influence some performance measure or quality characteristic of the product or process. The input variables are sometimes called *independent variables*, and are subject to the control of the engineer or scientist, at least for the purposes of the experiment. The performance measure or quality characteristic is called the *response*. The general objective of the RSM process is to find values for the input variables that yield a desired, often "optimal" response.

Since the form of the true response function *f* is unknown, it must be approximated. The successful use of RSM is critically dependent upon the experimenter's ability to develop a suitable approximation for *f*. A response surface is the geometric representation of a response function. [Hil98]

The ability to observe the response of a system is often skewed by variability and uncertainty. If repeated observations of the system are made at the same set of input conditions, the responses observed may vary from observation to observation because of: measurement errors, variability in the "experimental material", or the influence of other variables not accounted for.

After observing the response at different sets of values of the input variables, an attempt is made to use this information to develop a parsimonious approximation to the response function referred to as the empirical model. RSM comprises a set of statistical and mathematical techniques for empirical model building and exploitation that encompasses [Hil98]:

1. Designing a series of experiments that will yield adequate and reliable measurements of the response(s) of interest in a region of interest.
2. Analyzing the results of those experiments to determine an empirical model that best fits the data collected
3. Searching for the optimal settings of the input variables that produce a desired response

These techniques include:

1. *Designed Experiments* – an experimental process of inducing purposeful changes in the input variables in order to observe and model the changes in the response
2. *Regression Analysis* – statistical techniques used to model the response as a linear combination of various forms of the input variables and their interactions
3. *Steepest Ascent* – a gradient search technique that helps us to "scale the heights" of the response surface

This research is targeted at demonstrting that constructive induction provides a suitable learning mechanism for the virus detector system of an individual computer system. Based on the empirical analysis of MERCURY's results, presented above, the hypothesis that constructive induction provides a suitable learning mechanism for the virus detector system of an individual computer system can not be accepted or rejected.

The RSM methodology can be applied to the constructive induction approach to the virus detection domain. The output from the testing of MERCURY can be manipulated using statistical and mathematical techniques, in order to possibly *improve and optimize the process* of virus detection using the constructive induction approach. The input variables influencing the performance and quality of the virus detection are numerous. These independent variables are listed in the following table; each is defined by its name, type, and range.

Table 22 -- RSM input variables

Input Variable	**Type**	**Range**
Test Case Number	Nominal	1 - 11
Number of Hypotheses Generated	Continuous	1 – 1,000,000
Method used for Hypothesis Generation	Nominal	Chunking, EOSW, Sliding Window
Operator used for Hypothesis Generation	Nominal	NONE, AND, OR, XOR, BEFORE, DISTANCE
Hypothesis Label	Nominal	Self or Nonself
Induction Type	Nominal	Selective or Constructive

Each of these input variables could potentially contribute, in varying degrees, to the quality of the response, which in our case is the "goodness" of constructive induction applied to the virus detection domain. Specifically, this "goodness" is measured through two scores of power and purity for each class of hypotheses. A class of hypotheses are those generated by the same combination of constructive operators and selection rule, and yielding the same scores for power and purity. Each hypothesis within a class might represent a different group of features; however, this research focused upon the generation method. Therefore, these two scores will be utilized as the response variables intended for optimization. The response variables are defined in the following table, by name, type, and range. The general objective of this RSM process is to find values for the input variables that yield an "optimal" response for power and purity.

Table 23 -- RSM response variables

Response Variable	**Type**	**Range**
Power	Continuous	0.0 - 1.0
Purity	Continuous	0.0 - 1.0

The data produced by MERCURY contains a certain amount of variance, some of which can be explained. The output generated by MERCURY exhibits a high degree of variability *between* test cases and a low degree of variability *within* test cases. The variability between test cases is most evident between test cases 1 through 8 and between the "group" 1 through 8 and the "group" 9 through 11. The heterogeneity between test cases was primarily driven by the objectives of each test case. Test cases 1 through 8 represent contrived, laboratory test cases, whereas 9 through 11 represent the operational test cases, with byte patterns extracted from "real-world" non-infected files and "real-world" virus files. The low variability within test cases could be explained by the small size of individual test cases, the small size of extracted byte patterns, and the method of contriving the test cases in order to test the functionality of certain generation methods and constructive operators.

MERCURY was tested on 11 test cases, using different sets of values of the input variables. However, the process optimization techniques will only be applied to the operational test cases. The laboratory test cases were only used to verify and validate

MERCURY, which renders them unsuitable for optimization. The primary RSM technique used in this dimension of analyses is *regression analysis*, which attempts to model the response as a linear combination of various forms of the input variables and their interactions. These analyses will examine the output for the operational test cases to determine the *existence* of a parsimonious, empirical model. The RSM techniques will also determine which input variables have the greatest effect on the responses of power and purity, and determine the optimal "settings" of the input variables which produced the optimal responses.

The following sections present the findings of the RSM study. All analyses were obtained using the JMP Statistical Analysis Program. The first section describes the distribution characteristics of the data set, and makes provisions for their analyses.

6.7.1. Analyzing the Distribution of Data

A normality test was conducted on the test cases, using the Shapiro Wilk Test for Normality. These tests concluded that the data set was not normally distributed. If the p-value is less than 0.05 or some other alpha, conclusions of a non-normal distribution can be made. The following table, showing the results of the normality test, which supports the conclusion of non-normality. The purity scores were all 1.0 for the operational test cases; therefore, they were not tested or analyzed.

Table 24 -- Test for Normality

Shapiro-Wilk W Test - Test for Normality			
Test Case Group	**Response**	**W**	**p-value**
9 -11	Power	0.785653	0.0000
	Purity	-	-

Based on these results, nonparametric tests were used for this data. These types of tests do not depend upon a normal distribution of data; nor do they depend on the assumption that the population from which the sample was taken is normal [All90]. The assumptions for nonparametric tests are that the samples are independent from each other, their variances are constant, and their residuals are normally distributed. Nonparametric methods are often computationally simpler and easier to understand; however, they also have some disadvantages. One disadvantage is that they tend to be less sensitive than their parametric counterparts and thus require stronger evidence to reject a null hypothesis [All90].

The first assumption of independence is satisfied, based upon the deterministic qualities of MERCURY. The second assumption of constant variance is partially satisfied. Based on the results presented in the following three tables, variance across methods and labels is constant, though variance across operators is not constant. These conclusions are based on at least one test for each group, where the probabilities greater than 0.05 showed constant variance, and probabilities less than 0.05 showed non-constant variance.

Table 25 -- Tests that the Variances are Equal across Operators

Test	F Ratio	DF Num	DF Den	Prob>F
O'Brien[.5]	8.5902	5	162	<.0001
Brown-Forsythe	3.2398	5	162	0.0081
Levene	6.0585	5	162	<.0001
Bartlett	6.1502	5	?	<.0001

Table 26 -- Tests that the Variances are Equal across Methods

Test	F Ratio	DF Num	DF Den	Prob>F
O'Brien[.5]	0.9712	2	165	0.3808
Brown-Forsythe	1.4102	2	165	0.2470
Levene	1.7973	2	165	0.1690
Bartlett	1.2445	2	?	0.2881

Table 27 -- Tests that the Variances are Equal across Labels

Test	F Ratio	DF Num	DF Den	Prob>F
O'Brien[.5]	38.6499	1	166	<.0001
Brown-Forsythe	3.7111	1	166	0.0558
Levene	63.1145	1	166	<.0001
Bartlett	30.1749	1	?	<.0001

The third assumption of normally distributed residuals is not satisfied. The following figure shows the distribution of residuals. Based on the results of the Shapiro Wilk Test for Normality, with a p-value of <. 0001, these tests concluded that the residuals were not normally distributed. Although not all three assumptions for nonparametric testing were fully satisfied, the analyses *will* use this form of testing. Since the data from the test cases are nominally scaled, the applicability of variance and normality could be lessened.

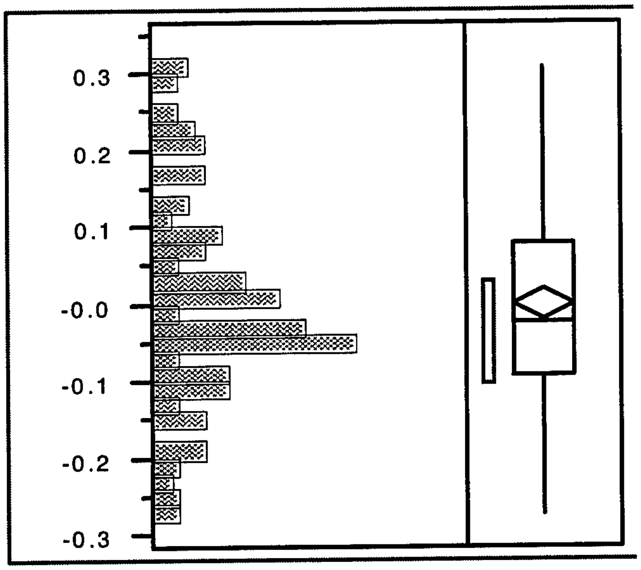

Figure 47 -- Distribution of Residuals

6.7.2. Analyses on Test Cases 9 through 11

6.7.2.1. Preliminary Analyses

The analysis on the operational test cases used the input variables Test Case Number, Number of Hypotheses Generated, Method used for Hypothesis Generation, Operator used for Hypothesis Generation, and Hypothesis Label as factors affecting the responses of power. There are no tests for purity, since all the hypotheses had a score of 1.0 across all test cases.

For each of these variables, a pictorial representation of the data is presented, showing the range, mean, and distribution of the power scores. The tables following each figure show the results of the Wilcoxon / Kruskal-Wallis Test, a nonparametric tool that tests whether group medians are the same across all groups. The usual analysis of variance assumption of normality is not made. "Prob>ChiSQ" is the probability of obtaining by chance alone a chi-square value larger than the one calculated if, in reality, the distributions across factor levels are centered at the same location [All90]. Observed probabilities of 0.05 or less are often considered as evidence that the distributions across input variable levels are not centered at the same location. Assumptions of the Kruskal-Wallis Test are that the populations have identical shape and variation, and that they are not normally distributed.

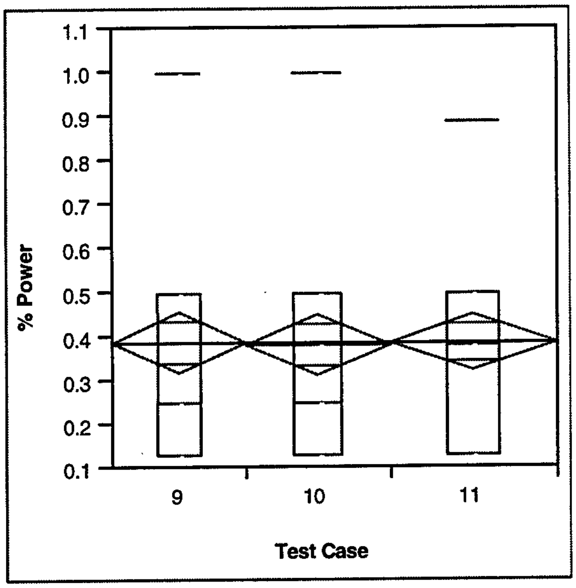

Figure 48 -- Pictorial View of Power across Test Cases

Table 28 – Wilcoxon/Kruskal-Wallis for Differences in Power across Test Cases

Wilcoxon / Kruskal-Wallis Tests (Rank Sums)				
Level	Count	Score Sum	Score Mean	(Mean-Mean0)/Std0
9	51	4307	84.4510	-0.007
10	55	4580.5	83.2818	-0.232
11	62	5308.5	85.6210	0.234

1-way Test, Chi-Square Approximation		
ChiSquare	DF	Prob>ChiSq
0.0721	2	0.9646

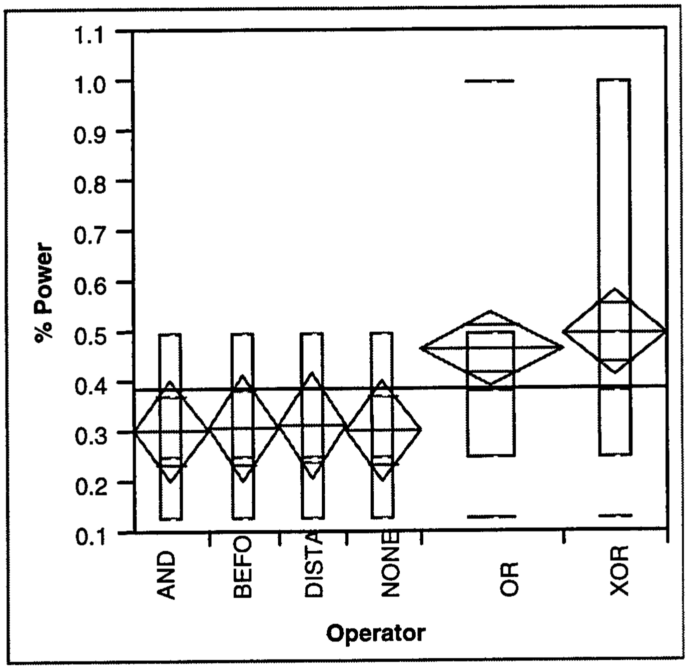

Figure 49 -- Pictorial View of Power across Operators

Table 29 – Wilcoxon/Kruskal-Wallis for Differences in Power across Operators

Wilcoxon / Kruskal-Wallis Tests (Rank Sums)				
Level	Count	Score Sum	Score Mean	(Mean-Mean0)/Std0
AND	24	1762	73.4167	-1.244
BEFORE	22	1628	74.0000	-1.120
DISTANCE	21	1561	74.3333	-1.056
NONE	24	1762	73.4167	-1.244
OR	45	4346	96.5778	2.010
XOR	32	3137	98.0313	1.805

1-way Test, Chi-Square Approximation		
ChiSquare	DF	Prob>ChiSq
10.3443	5	0.0660

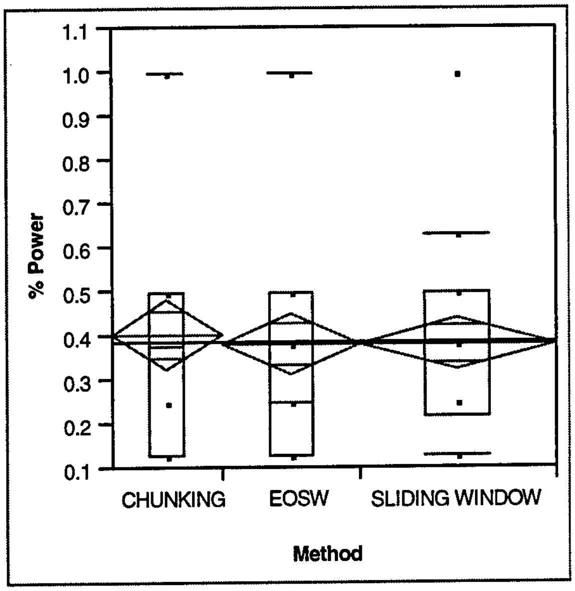

Figure 50 -- Pictorial View of Power across Methods

Table 30 – Wilcoxon/Kruskal-Wallis for Differences in Power across Methods

Wilcoxon / Kruskal-Wallis Tests (Rank Sums)				
Level	Count	Score Sum	Score Mean	(Mean-Mean0)/Std0
CHUNKING	42	3586.5	85.3929	0.140
EOSW	52	4267.5	82.0673	-0.447
SLIDING_WINDOW	74	6342	85.7027	0.292

1-way Test, Chi-Square Approximation		
ChiSquare	DF	Prob>ChiSq
0.2023	2	0.9038

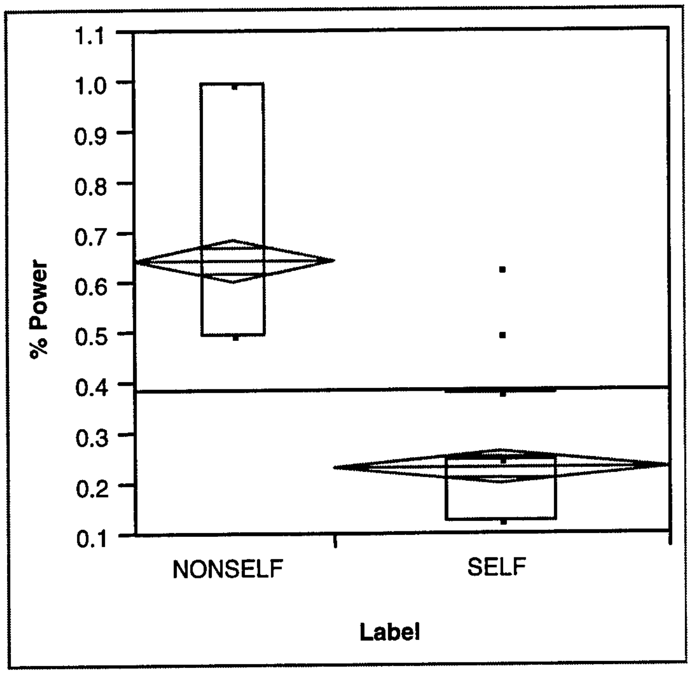

Figure 51 -- Pictorial View of Power across Labels

Table 31 – Wilcoxon/Kruskal-Wallis for Differences in Power across Labels

Wilcoxon / Kruskal-Wallis Tests (Rank Sums)				
Level	**Count**	**Score Sum**	**Score Mean**	**(Mean-Mean0)/Std0**
NONSELF	63	8406	133.429	10.435
SELF	105	5790	55.143	-10.435

1-way Test, Chi-Square Approximation		
ChiSquare	**DF**	**Prob>ChiSq**
108.9314	1	<.0001

Based on the results from the figures and tables above, the following conclusions can be drawn. The power scores of each test case are not significantly different from each other, shown by a probability score of 0.9646. These are the expected results, since the examples in these test cases were generated from "real world" files.

The power scores generated by the different operators are statistically different, proven by the probability value of 0.0660. In order to accept the conclusion of difference, an alpha of 0.10 is required, as opposed to 0.05, which provides less confidence in these results. This test is significant because it proves that at least one operator or group of operators provided a better measure of power than the other operators. By observation, it could be concluded that OR and XOR are better operators, based on their high power scores of 96.5778 and 98.0313, respectively. These scores were obtained from Table 29. By performing additional tests, it can be shown *which* operator is different. Isolating the four variables believed to be similar, AND, NONE, BEFORE and DISTANCE, the test below confirms they are statistically the same.

Table 32 – Wilcoxon/Kruskal-Wallis for Differences in Power across FOUR OPERATORS

<table>
<tr><th colspan="5">Wilcoxon / Kruskal-Wallis Tests (Rank Sums)</th></tr>
<tr><th>Level</th><th>Count</th><th>Score Sum</th><th>Score Mean</th><th>(Mean-Mean0)/Std0</th></tr>
<tr><td>AND_OP</td><td>24</td><td>1101.5</td><td>45.8958</td><td>-0.019</td></tr>
<tr><td>BEFORE</td><td>22</td><td>1013.5</td><td>46.0682</td><td>0.010</td></tr>
<tr><td>DISTANCE</td><td>21</td><td>969.5</td><td>46.1667</td><td>0.030</td></tr>
<tr><td>NONE</td><td>24</td><td>1101.5</td><td>45.8958</td><td>-0.019</td></tr>
<tr><th colspan="5">1-way Test, Chi-Square Approximation</th></tr>
<tr><th colspan="2">ChiSquare</th><th>DF</th><th colspan="2">Prob>ChiSq</th></tr>
<tr><td colspan="2">0.0020</td><td>3</td><td colspan="2">1.0000</td></tr>
</table>

If the OR operator is added to this group of four, the similarity of the OR operator with the remaining operators can be tested. Running the same tests yielded a probability score of 0.1224, concluding the operators are similar, using an alpha of 0.05. Similar results were obtained when XOR was added to the group of four operators, giving a probability value of 0.1751. Finally, XOR and OR were compared, resulting in a probability score of 0.8867. Though not statistically proven, some conclusions can be drawn by observation. The XOR and OR operators likely yield a higher power score than the other operators. This information may be useful when ordering the operators for use in a constructive induction approach to virus detection. If the more useful operators are used first, the computational complexity of the entire operation may be improved.

Since "NONE" was used as the operator for the selection method, the implications are that selection did not produce a power score significantly different from the power scores produced when using a constructive operator. This test should not be used alone to discount the effects of construction over selection. Since the sample sizes were so small, and sensitivity could be lost with nonparametric tests, further investigations would be needed to conclude the effectiveness of this learning method.

The probability value of the Chi-square test for the label test is <.0001, implying a difference exists in the score for power due to the label of self or nonself. Hypotheses generated to detect nonself have a statistically better power score than those generated to detect self.

In both cases of power and purity, results could be skewed by the fact that there was a 4 to 1 ratio of self files to nonself files in the example set. The possible values of power for nonself were {0%, 50%, 100%}; the possible values of power for self were {0%, 12.5%, 25%, 37.5%, 50%, 62.5%, 75%, 87.5%, 100%}. The disparity between these two sets could have artificially inflated the effects of the hypothesis label on power.

Since purity is a measure of how correctly the hypothesis classifies the examples, this score could also be affected by the difference.

6.7.2.2. Regression Analyses

To confirm the findings from the nonparametric tests, a regression test was run, which tested the "effects" of the input variables on the specified response of power. Though the data set was not normally distributed, these regression tests are robust enough to allow for non-normality. The results of these tests are derived from a model that exhibited a "good fit."

There are three primary measures that assess a fitted model: MSE, R^2 and F. A "good" model will be significant, as indicated by a "large" value of F, exhibit a "small" error component MSE and explain most of the variation in the responses by having a "large" R^2.

Table 33 – Analysis of Variance (ANOVA) Table for the Power Model

Source	**DF**	**Sum of Squares**	**Mean Square**	**F Ratio**
Model	11	8.611177	0.782834	**42.9359**
Error	156	2.844289	**0.018233**	Prob>F
C Total	167	11.455466		<.0001
Label	1	6.2961494	345.3233	<.0001

The Analysis of Variance (ANOVA) table above shows the significance of this model through the "large" F Ratio of 42.9359. This indicates at least one of the input variables is contributing significantly to the model. The Mean Square Error (MSE) provides a measure of the variability within the residuals and provides a measure of how well the fitted responses match those observed. The table above shows the "small" error value of 0.018. In the following table, the R^2 value of 0.75 represents the proportion of the variability within the observed responses that can be explained or accounted for by the model. Additionally, developing a "best" model usually involves finding a model that has the above characteristics and is "parsimonious", meaning it involves the fewest parameters. It is difficult to just delete parameters, this process is often facilitated by testing the addition or removal of the variables. [Hil98]

The model derived here is a "good" model, meaning it can be used to gain insightful knowledge into the important factors affecting the power response. This could be very useful in the application of constructive induction to virus detection. As stated previously, the computational complexity of this learning method is extreme. A method, such as regression can help pinpoint and guide the learning process by identifying early in the process the important variables and the settings needed to optimize the desired responses of high power and high purity.

Table 34 depicts the results of the effect test for the model. This test shows the impact of variables on the response of power. The F test is used to determine if all settings of a variable have the same effect. The null hypothesis states all settings have equal means, and the alternative hypothesis states that at least two means are different. The variable settings with "large" F ratios and "small" probabilities, of less than alpha, reject the null hypothesis. Using an alpha of 0.05, the most significant effects are the operator, method, and label. This indicates that at least one setting within each variable has a different mean than the others. Further analyses, using the parameter estimates' t tests indicated which settings were the most significant.

Table 34 – Effect Test for the Power Model

Source	Nparm	DF	Sum of Squares	F Ratio	Prob>F
Test Case	2	2	0.018	0.519	0.5957
Operator	5	5	1.783	19.56	**<.0001**
Method	2	2	0.145	3.98	**0.0204**
Label	1	1	6.296	345.32	**<.0001**

Table 35 shows the parameter estimates for the variables and their settings, and their tests of significance. Since the input variables are nominal, each term symbolizes a particular variable setting compared to the rest of the settings for that variable [SAS95]. For example, the term "Operator[AND-XOR]" symbolizes the effect of the AND operator compared to the group containing the rest of the operators. A t test is performed which indicates the significance of that particular variable setting. This tests whether the setting of a particular variable is statistically different from the other settings of that variable. The null hypothesis states all settings are equal, and the alternative hypothesis states that a particular setting is not equal. The variable settings with "large" t ratios and "small" probabilities, of less than alpha, reject the null hypothesis. Using an alpha of 0.05, all of the operators and both of the label variables produced high t ratios, meaning these terms are significantly different from each other. With an alpha of 0.10, the method of chunking becomes significantly different than the other two generation methods. The other generation methods and the specific test cases do not appear to be significantly different from each other.

Table 35 – Regression Information for the Power Model

Summary of Fit	
RSquare	**0.751709**
RSquare Adj	0.734201
Root Mean Square Error	0.135028
Mean of Response	0.385893
Observations (or Sum Wgts)	168

Parameter Estimates				
Term	**Estimate**	**Std Error**	**t Ratio**	**Prob>|t|**
Intercept	0.4202101	0.011644	36.09	<.0001
Test Case[9-11]	-0.007323	0.015182	-0.48	0.6303
Test Case[10-11]	-0.007414	0.014863	-0.50	0.6186
Operator[AND-XOR]	-0.064471	0.025034	-2.58	**0.0109**
Operator[BEFORE-XOR]	-0.07446	0.025924	-2.87	**0.0046**
Operator[DISTANC-XOR]	-0.078363	0.026439	-2.96	**0.0035**
Operator[NONE-XOR]	-0.070195	0.025241	-2.78	**0.0061**
Operator[OR-XOR]	0.1033423	0.020025	5.16	**<.0001**
Method[CHUNKIN-SLIDING]	-0.031237	0.016629	-1.88	**0.0622**
Method[EO_SLID-SLIDING]	-0.008616	0.015363	-0.56	0.5757
Label[NONSELF-SELF]	0.2101124	0.011307	18.58	**<.0001**

Breaking the model down even further, additional analyses can be conducted. The least squares means are predicted values from the specified model across the levels of each variable setting, where the other variables are controlled by being set to neutral values. The least squares means are the values to examine to see which levels produce higher responses from power, holding the other variables constant [SAS95]. The following three tables show the least squares mean scores for the operators, methods, and labels. XOR and OR appear to give the largest power scores of all operators, sliding window appears to give the highest power score of all methods, and hypotheses with the nonself label appear to have higher power scores.

Table 36 – Least Squares Means for the Operators

Least Squares Means			
Level	**Least Sq Mean**	**Std Error**	**Mean**
AND	0.342110	0.027985	0.30416
BEFORE	0.332121	0.029111	0.30909
DISTANCE	0.328219	0.029712	0.31190
NONE	0.336387	0.028284	0.30416
OR	**0.509924**	0.020420	0.46555
XOR	**0.590728**	0.024710	0.49781
Effect Test			
Sum of Squares	**F Ratio**	**DF**	**Prob>F**
10.3443	19.5647	5	**<.0001**

Table 37 – Least Squares Means for the Methods

Least Squares Means			
Level	**Least Sq Mean**	**Std Error**	**Mean**
CHUNKING	0.375345	0.021602	0.40357
EOSW	0.397965	0.019127	0.37942
SLIDING_WINDOW	**0.446435**	0.016673	0.38040
Effect Test			
Sum of Squares	**F Ratio**	**DF**	**Prob>F**
0.14544740	3.9887	2	**0.0204**

Table 38 – Least Squares Means for the Labels

Least Squares Means			
Level	**Least Sq Mean**	**Std Error**	**Mean**
NONSELF	**0.616694**	0.017767	0.64285
SELF	0.196469	0.014186	0.23171
Effect Test			
Sum of Squares	**F Ratio**	**DF**	**Prob>F**
6.2961494	345.3233	1	**<.0001**

6.7.3. Utilizing Process Optimization

In order to show the useful application of the process optimization techniques discusses above, a case study of their utilization was simulated. Since insight was gained about the effects of different operators on the response of power, this knowledge was included in a simulation of MERCURY. This simulation *only* used the OR and XOR operators to construct new hypotheses. The results from this simulation are presented below.

Power of construction before optimization	**Power of construction after optimization**
22.6% self	25% self
66.9% nonself	83% nonself
31.5% overall	36.3% overall

The results show an increase of average power scores for both self and nonself hypotheses. The overall weighted average also increased. Additionally, this optimized simulation produced 18.4% fewer hypotheses. By knowing the effects of the OR and XOR operators, they can be used more effectively in the inductive process. They produced a better average, and decreased some of the computational growth. Though this decrease is small, it represents only a small tidbit of *a priori* knowledge. If combined with other pieces of knowledge, the computational growth could be reduced even further.

6.7.4. RSM Conclusions

The results of these tests provided evidence that statistical methods, like the ones used in RSM processes, can provide empirical analysis and knowledge that could make the constructive induction learning process more efficient. Although little insight could be gleaned from the first group of tests cases, the second group of test cases was able to demonstrate the capabilities of process optimization technique. Since the main disadvantage of constructive induction is its computational explosion, these results provide mathematically-based methods that could decrease its computational complexity, by providing knowledge about the problem domain *a priori*.

Based on the knowledge obtained from the last three test cases, future runs of MERCURY could be optimized by utilizing the sliding window method before other methods. In addition, construction could begin with the XOR and OR operators, followed by the others, if necessary. This could reduce the time and space explosions explained in previous sections of this chapter.

6.8. Summary

This chapter presented the results of various test scenarios utilizing MERCURY. These results were analyzed by the five performance dimensions of time, space, power and purity, coverage and process optimization. Time and space were both recognized as potential downsides to MERCURY; however, several optimization methods to future code iterations and algorithms that could reduce these effects are presented in Chapter Seven. Power and purity scores, in general, were shown to increase between selection and construction, possibly indicating useful selection methods, operators, or constructive rules. Although a statistically significant improvement between selection and construction using the optimization techniques of RSM was not shown, other important information was obtained through this analysis. These techniques were recognized as potential "guidelines" for increasing the performance of a constructive induction learning engine. RSM could provide the virus detection programmer *a priori* knowledge, resulting in a better detection system. Further conclusions from these analyses are presented in Chapter Seven.

7. Conclusions

7.1. Research Overview

This research integrated four different domains: computer virus detection, human immunology, computer immunology and constructive induction. The goal of this research was three-fold. First, a computer health model was defined that could possibly improve the current "global" approach to computer viruses. This health model was based on the public health system, and provided a high level view of a Computer Health System. Second, a computer immune model was defined that could possibly improve the current "local" approach to computer virus detection. This detection model was based on the human immune system, and provided a high level view of an individual computer immune system. Third, a detection model was developed, represented by the prototype MERCURY. This model utilized the machine learning concept of constructive induction to capture the human immune characteristic of self-adaptation. The work accomplished as part of this investigation tested the primary hypotheses.

7.2. Research Hypotheses

Primary Hypotheses

1. The public health system is a useful model for a Computer Health System for the global protection of computer system against viruses
2. The human immune system is a useful model for a virus detection system on an individual computer system
3. Constructive induction provides a suitable learning mechanism for the virus detector system of an individual computer system

The first two objectives of this research were to test the first two hypotheses to determine if the public health system and the human immune system were useful models for a Computer Health System and computer immune system, respectively. To accomplish this objective, research in the areas of public health and human immunology was conducted. The requirements, objectives and components of the models were also evaluated. Both computer models are informal, explanatory models based on some essential qualities of their respective systems. Due to the models' informalities, though, not all of their aspects were explicitly stated.

Though the first two objectives were not *formally* tested, the first two hypotheses can be supported. The Computer Health System was derived by analogy from an effective system in an applicable domain. The computer immune system was also derived by analogy, and its main functions of detection, adaptation and memory were translated into the design of the prototype, MERCURY, and abstractly demonstrated in its implementation.

The third objective of this research was to empirically test the third hypothesis. This objective investigated whether constructive induction was suitable for virus detection in a computer immune system. Testing was conducted utilizing MERCURY. While MERCURY captures the essence of constructive induction, it does not fully employ all the characteristics of a complete inductive engine. The analyses supported the third hypothesis by failing to reject it, and by showing empirical evidence that construction improved classification. In other words, MERCURY was not able to validate, or refute, that constructive induction *definitively* provides a suitable learning mechanism for the virus detector system of an individual computer system.

The results of these tests *did* provide empirical evidence and analytical knowledge that could make the learning process more efficient. Since the main disadvantage of constructive induction is its computational explosion, these results provided mathematically based methods which could decrease its computational complexity. These methods could improve the capabilities of a fully developed constructive induction based virus detector, by providing knowledge about the problem domain and the system parameters *a priori*. To confirm these findings, a process optimization simulation was conducted to demonstrate the effectiveness of *a priori* knowledge applied to the virus detection problem.

The third hypothesis was decomposed into smaller, more manageable, sub-hypotheses. The first sub-hypothesis was the Virus Feature Hypothesis:

Virus Feature Hypothesis

Byte patterns can be used as the basis of a constructive induction based computer virus detector.

Although current virus research stated byte patterns were useful features, it was necessary to ensure constructive induction did not decrease detection capabilities. To confirm this, MERCURY's learning component was programmed to extract, manipulate, and test byte patterns from various files. Testing concluded that the learning component, using features composed of byte patterns, was able to detect self and nonself files with varying degrees of accuracy. Therefore, it can be concluded that byte patterns can be used as the basis of a constructive induction based computer virus detector.

The second sub-hypothesis was the Constructive Operator Hypothesis:

Constructive Operator Hypothesis

Logical and spatial operators can be used for constructing new attributes for the computer virus detector.

Current virus research confirms the applicability of using relative and absolute locations of virus characteristics to detect an infected file. This hypothesis validated that the choice of operators combining these characteristics was adequate, better distinguishing between infected and uninfected data. To confirm this, MERCURY's learning component was programmed to manipulate the byte patterns from various files based on two types of operators, logical and spatial. Testing concluded that the learning component, using logical and spatial operators, was able to detect self and nonself files with varying degrees of accuracy. While the logical operators performed better in this system, spatial operators should not be discounted. Further testing of these spatial operators should be conducted. Therefore, it can be concluded that logical operators can be used for constructing new features for the computer virus detector. More research is required for spatial operators.

7.3. Research Implications

The results of analyzing MERCURY demonstrate an inherent lack of representational power of computer virus byte patterns using selective induction methods. Constructive induction provides new, potentially powerful, and often necessary representations. However, the results of this research confirmed constructive induction's main deficiency, the explosion in the number of hypotheses generated.

The effects of this deficiency can be improved by utilizing key pieces of knowledge to guide construction. Process optimization through statistical techniques, provides direct insight into these key pieces of knowledge. Many factors influence the computational explosion of this system. Some of these are: feature size, file size, number of selection methods, number of operators, number of constructions, and construction rules. Examples of some guidelines for improving the performance of constructive induction in a virus detector are: the ordering of the selection methods, the ordering of the operators, the sequence of selection and construction, the appropriate time to evaluate, and the appropriate hypotheses to evaluate. Knowledge about the virus domain, such as characteristics of typical viruses and regularities in the byte patterns, also provides guidance for effective construction. However, care must be taken to not constrict or oversimplify the problem.

7.4. Research Limitations

Several factors limited aspects of this research. Due to the broadness of the four research areas and the disparity of the concepts being intertwined, time was a limitation. As proven in the subsequent section discussing future research topics, this problem domain is without boundaries in a vast number of directions. This research focused on the detection component of an individual computer immune system as part of a larger Computer Health System. Since these computer immune models were informal models, the validity of the models is limited to common sense and intuition.

Other limitations to this system were caused by hardware configuration and software design constraints, such as processing speed and memory and limited data structures. In addition, due to the computational complexity of this learning method, the number of test cases, the number of generation methods, the number of operators, the number of construction rules, and the number of constructions were all limited.

The results of testing and the conclusions drawn from them were also limited. Due to the small file sizes and example set sizes, and the non-normal distribution of this particular data, conclusions can not be *absolutely* validated. On the other hand, the primary hypothesis for constructive induction applicability in the virus detection domain *can not be rejected.* There is not enough evidence to support the claim that constructive induction is the answer to virus detection. However, evidence suggest that through empirical analyses and statistical techniques, improvements can be made over current methods of virus detection

7.5. Future Research

7.5.1. Computer Immunology

- *Application of computer immune models to other domains.* The Computer Health System model and the computer immune model that were developed in this research can be utilized in other computer security domains. These models provide global scoped and locally driven protection for computer networks and individual computers. The applicability of these models to intrusion detection, change detection, and malicious user detection should be explored. An additional goal of this research should be the integration of the various domains through these models.
- *Model refinement.* This research provided an overview of the Computer Health System, as discussed in Chapter Three. Further model refinement is needed to expand components, specify interfaces between the global Computer Health System and the local computer immune model, and specifying the communication protocols between systems.

- *Expanding the computer immune model.* This research focused on specifying the detection component of the computer immune model. The system analysis, virus elimination, and file repair tasks need to further specification. Current antivirus techniques, discussed in Chapter Two, can be incorporated into the model to handle these tasks.
- *Other aspects of immune system models.* This research investigated the use of the human immune system's defenses against intracellular and extracellular infection as a means of detecting computer viruses. Certain immunological aspects were not fully addressed in the model specification, requiring future work. These areas include autoimmunity, allergy, B-cell and T-cell interaction, and the role of macrophages. Additionally, a new concept of immune system operation, the *danger theory*, should be investigated. This theory claims that the immune system does not work through detection of self and nonself, but rather through detection of dangerous nonself through a costimulatory signal produced by antigen presenting cells [Ric96, Pen96].

7.5.2. Machine Learning

- *Different forms of machine learning.* The use of constructive induction in this research was not the only from of machine learning available. As discussed in Section 2.5, three current forms of machine learning, neural networks, genetic algorithms, and intelligent agents should be studied for their applicability to this domain. These forms of learning should be aided by the empirical evidence that that was collected and utilized in this research.

- *User intervention.* The current structure of MERCURY does not allow the user to intervene into either the learning or virus detection processes. Future research should investigate means for correcting overlearning or underlearning by HEC, proactively adding self to the knowledge base, and handling unclassified files.
- *Constructive Induction Code Optimization.* Several aspects of the constructive induction process can be optimized based upon the results of this research.
    - *Data structure improvements.* The hypotheses are currently stored in a flat list structure. This structure is traversed several times to construct hypotheses and evaluation. The system could be improved if an index into the features of the hypotheses was maintained. With such an index, it would be possible to search through the example file once to determine coverage, power and purity. Coverage is currently determined independently of the score for power and purity.
    - *Inclusion of domain specific bias from either empirical evidence or antivirus researchers.* Very little virus specific bias is included in HEC. The empirical results of this investigation and other biases could be used to guide the choice of hypotheses used for

construction, the operators used to construct, and the selection of features from the files. Relaxing these biases would help determine their affect on learning. This research should attempt to reduce the average execution time and memory requirements.

    - *Extracting Signatures*. HEC does not attempt to find the minimal number of hypotheses necessary to classify the examples, with high power and purity. Determining the subset of hypotheses needed to classify self and nonself is needed. In order to determine this, a mapping between the examples and the classifying hypotheses are needed. Additionally, analyses of the "outlier hypotheses" could be conducted to determine if they possess highly effective characteristics.
    - *Testing the predictive capability.* Several tests exist for validating the machine learning process and determining the predictive capability of the detectors. These tests should be used inside HEC when evaluating candidates for signatures.

- *Future RSM applications.* Regression analysis is one response surface methodology (RSM) technique used to investigate the relationships between process input parameters and process results. Additional techniques could be used, such as *steepest ascent*, which allows the experimenter to "scale" the heights of a response surface, in order to find the optimal region. This "hill climbing" technique is especially useful when there is more than one response variable which needs to be optimized, or there are additional constraints on the system. *Design of experiments* can also be used on different system variables to improve the application of the RSM techniques. Using differing levels of feature size, file size, etc, can give more insight into which variables can be optimized.

7.5.3. Virus Detection

- *Different types of viral detection.* This research explored the use of bytes as features for a constructive induction based computer virus detector. Other antiviral techniques use heuristics and system call analysis to determine if a computer is infected. A study of MERCURY's techniques should be investigated to provide a multilayered defense for the computer immune system, akin to the multilayered defense provided by the innate and adaptive immune system of the human body.
- *Different types of viruses.* MERCURY is designed to detect file infector viruses. Boot sector, polymorphic, and stealth viruses should be researched to determine methodologies to integrate detectors for these viruses with MERCURY.
- *Dynamic virus scanning.* MERCURY currently detects only when invoked by the user. In order to detect viruses as they infect a file, dynamic virus scanning is needed.

This study began by building two models: a Computer Health System and an individual computer immune system. Once the modeling for the two systems was complete, the prototype of MERCURY was designed and developed to capture the essence of the individual computer immune system. The overall results provided an analysis of constructive induction approach applied to the virus detection domain, as well as areas for optimizing this learning method and reducing its computational complexity.

This research recognized specific areas of improvement in machine learning that could be applied to current methods of virus detection, in order to improve performance. It also presented the incorporation of this constructive induction component into an individual computer's immune system, and further incorporated this system into an overall global picture of computer health.

Appendix A -- Source Code

The source code for MERCURY is not included as part of this document. Those interested in obtaining a copy of the source code should direct their requests to:

Dr. Gregg Gunsch

AFIT/ENG
2950 P Street
WPAFB, OH 45433-7765

gregg.gunsch@afit.af.mil

Bibliography

[AAV97] Dr. Solomon's Virus Central "All About Viruses 97." Available Online at http://www.drsolomon.com/vircen/vanalyse/va002.html

[ABKS94] Aytug, Haldun, Siddhartha Bhattacharyya, Gary J. Koehler and Jane L. Snowdon. "A Review of Machine Learning in Scheduling." *IEEE Transactions on Engineering Management*, 41(2): 165-171, May 1994.

[All90] Allen, Arnold O. *Probability, Statistics, and Queuing Theory With Computer Science Applications.* Second Edition. Academic Press: Boston, 1990.

[Ayr96] Ayres, Stephen M., M.D. *Health Care in the United States.* Chicago: American Library Association, 1996.

[BlF90] Blanchard, Benjamin S. and Wolter J. Fabrycky. *Systems Engineering and Analysis.* New Jersey: Prentice Hall, 1990.

[Boe88] Boehm, Barry W. "A Spiral Model of Software Development and Enhancement." *IEEE Computer*, pgs 61-72, May 1988.

[BSL96] Benjamini, Eli, Geoffrey Sunshine and Sidney Leskowitz. *Immunology: A Short Course.* New York: Wiley-Liss, 1996.

[Cas97] Casti, John L., *Would-be Worlds.* New York: John Wiley & Sons, 1997.

[Che97] Chess, David. "The Future of Viruses on the Internet." In *7th Virus Bulletin International Conference.* Abingdon, England: Virus Bulletin, 1997.

[CLR90] Cormen, Thomas H, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* Cambridge: MIT Press, 1990.

[DeB92] DeRaedt, Luc and Maurice Bruynooghe. "Interactive Concept-Learning and Constructive Inductance by Analogy." *Machine Learning*, 9:107-150, 1992.

[DFH96] D'heseller, Patrik, Stephanie Forrest, and Paul Helman. "An Immunological Approach to Change Detection: Algorithms, Analysis and Implications." *Proceedings of the 1996 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, 1996.

[DiM81] Dietterich, Thomas G and Ryszard S. Michalski. "Inductive Learning of Structural Descriptions: Evaluation Criteria and Comparative Review of Selected Methods." *Artificial Intelligence*, 16:257-294, 1981.

[Elg96] Elgert, Klaus D. *Immunology: Understanding the Immune System.* New York: Wiley-Liss, 1996.

[FHS97] Forrest, Stephanie, Steven A. Hofmeyr, and Anil Somayaji. "Computer immunology." *Communications of the ACM*, 40(10):88 - 96, October 1997.

[FHSL96] Forrest, Stephanie, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. "A Sense of Self for Unix Processes." *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 120-128, 1996.

[FJSP93] Forrest, Stephanie, Brenda Javornik, Robert E. Smith and Alan S. Perelson. "Using Genetic Algorithms to Explore Pattern Recognition in the Immune System." 1993.

[Gun91] Gunsch, Gregg H. *Opportunistic Constructive Inductance: Using Fragments of Domain Knowledge to Guide Construction.* PhD thesis, University of Illinois at Urbana-Champaign, 1991.

[Guy81] Guyton, Arthur C. *Textbook of Medical Physiology*. Philadephia: Saunders, 1981.

[Hau87] Haussler, David. "Bias, Version Spaces and Valiant's Learning Framework." In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 324-336, Irvine, CA, June 1987.

[HFS97] Hofmeyr, Steven A., Stephanie Forrest, and Anil Somayaji. "Intrusion Detection using Sequrnces of System Calls." Department of Computer Science, University of New Mexico, December 1997.

[Hil98] Hill, Raymond, Maj, USAF. "Overview – What is RSM?" Class notes from *OPER 683 Response Surface Methodology.* Summer 1998.

[IBM98] IBM. "IBM and Symantec Combine Forces." IBM Press Release, May 1998.

[Jan93] Janeway, Charles A. Jr. "How the Immune System Recognizes Invaders." *Scientific American*, pages 73-79, September 1993.

[Jon92] Jonas, Steven, M.D. *An Introduction to the US Heath Care System.* 3rd Edition. New York: Springer Publishing Company, 1992.

[Kas99] Kaspersky, Eugene. "Viral Analysis Texts: Web version." Metropolitan Network BBS, Inc, 1999.

[KeA94] Kephart, Jeffrey O. and William C. Arnold. "Automatic Extraction of Computer Virus Signatures." In R. Ford, editor, *4th Virus Bulletin International Conference*, pages 179-194. Abingdon, England: Virus Bulletin, 1994.

[Kep94] Kephart, Jeffrey O. "A Biologically Inspired Immune System for Computers." Rodney A. Brooks and Pattie Maes, editors, *Artificial Life IV, Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pp. 130-139. Cambridge, Mass: MIT Press, 1994.

[KSCW97] Kephart, Jeffrey O., Greogry B. Sorkin, David M. Chess, and Steve R. White. "Fighting Computer Viruses." *ScientificAmerican*, pp. 88-93, November 1997.

[KSSW97] Kephart, Jeffrey O., Gregory B. Sorkin, Morton Swimmer, and Steve R. White. "Blueprint for a Computer Immune System." In Proceedings of the 7th Virus Bulletin International Conference. Abingdon, England: Virus Bulletin, 1997.

[LaS95] Langley, Pat and Herbert A. Simon. "Applications of Machine Learning and Rule Induction." *Communications of the ACM*, 38(11):55-64, November 1995.

[Lud98] Ludwig, Mark. *The Giant Black Book of Computer Viruses*, 2nd Edition. Arizona: American Eagle Publications, 1998.

[LVM98] Lamont, Gary B., David A. Van Veldhuizen, and Robert E Marmelstein, *A Distributed Architecture for a Self-Adaptive Computer Virus Immune System.* White paper. Air Force Institute of Technology, 1998.

[MaM95] Maloof, Marcus A. and Ryszard S. Michalski. "A Partial-Memory

Incremental Learning Methodology and its Application to Intrusion Detection." In Proceedings of the 7th *IEEE International Conference on Tools with Artificial Intelligence*, Herndon, VA, 1995.

[MVL98] Marmelstein, Robert E., David A. Van Veldhuizen, and Gary B. Lamont. *Modeling & Analysis of Computer Immune Systems Using Evolutionary Algorithms*. White paper, Air Force Institute of Technology, February 1998.

[NCSA96] NCSA Virus Lab, "Frequently Asked Questions (FAQ) ¼." pp. 7-11, November 29, 1996.

[Nos93] Nossal, Sir Gustav J. V. "Life, Death and the Immune System." *Scientific American*, pages 53-62, September 1993.

[OUSD96] Office of the Under Secretary of Defense for Acquisition and Technology. "Information Warfare – Defense: Appendix D Organizational Models." *Report of the Defense Science Board Task Force on Information Warfare – Defense (IW-D)*. 1996. Available Online at http://jya.com/iwd-d.htm.

[Pen96] Pennisi, Elizabeth. "Tetering on the Brink of Danger." *Science*, pages 1665-1667, 22 March 1996.

[Pro92] Provost, Foster John. *Policies for the Selection of Bias in Inductive Machine Learning*. PhD thesis, University of Pittsburgh, 1992.

[Qui86] Quinlan, J.R. "Induction of Decision Trees." *Machine Learning*, I:81-106, 1986.

[RBM98] Roitt, Ivan, Jonathan Brostoff and David Male. *Immunology*. Fifth Edition. London: Mosby, 1998.

[RBP91] Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. New Jersey: Prentice-Hall, 1991.

[Ren90] Rendell, Larry. "Learning Hard Concepts." *Computational Intelligence*. Vol 6, No. 4, 1990.

[Ric96] Richardson, Sarah. "The End of Self." *Discover*, 17(4): 80-88, April 1996.

[SAS95] SAS Institute Inc. *JMP Statistics and Graphics Guide*. SAS: Cary, NC, 1995.

[ShG96] Shaw, Mary, and David Garlan. *Software Architecture*. New Jersey: Prentice-Hall, 1996.

[Shr96] Shrobe, Howard. "ARPATech '96 Information Survivability Briefing." *Presented at the ARPATech '96 Systems and Technology Symposium*. Atlanta, Georgia, May 1996.

[Ste93] Steinman, Lawrence. "Autimmune Disease." *Scientific American*, pp. 107-114, September 1993.

[TKS96] Tesauro, Gerald, Jeffrey O. Kephart and Gregory B. Sorkin. "Neural Networks for Computer Virus Recognition." *IEEE Expert*, 11(4):5-6, 1996.

[Tur97] Turnock, Bernard J. *Public Health: What It Is and How It Works*. Maryland: Aspen Publishers, 1997.

[WCC89] White, Steve R., Jimmy Kuo Chengi and David M. Chess. "Coping with Computer Viruses and Related Problems." IBM Thomas J. Watson

Research Center, Research Report Number RC 14405, pp. 1-8, January 30, 1989.

[WeC93] Weissman, Irving L. and Max D. Cooper. "How the Immune System Develops." *Scientific American*, pages 65-71, September 1993.

[WFP96] White, Gregory B., Eric A. Fisch and Udo W. Pooch. Computer System and Network Security. New York: CRC Press, 1996.

[Win92] Winston, Patrick Henry. *Artificial Intelligence*. New York: Addison-Wesley, 1992.

[WnM94] Wnek, Janusz and Ryszard S. Michalski. "Hypothesis-Driven Constructive Induction in *AQ17-HCI*." *Machine Learning*, 14:139-168, 1994.

[Zin96] Zinkernagel, Rolf M. "Immunolgy Taught by Viruses." *Science*, 271:173-177, 1996.

Vitas

PII Redacted

Captain Cardinale, [redacted], graduated from the George Washington University in 1994. Prior to coming to AFIT, she had several duties in the communications and information career field. Captain Cardinale's first assignment was to the AETC Training Support Squadron, Randolph AFB, Texas, where she developed computer-based courseware used in undergraduate flying training. In 1996, she became the chief of the network administration section for the squadron. Following graduation from AFIT, Captain Cardinale will be assigned to the Air Force Technical Applications Center (AFTAC) at Patrick AFB, FL. She will be working in the Systems Integration Management Office, responsible for benchmarking AFTAC's current architecture requirements and building a roadmap for their future.

Lieutenant O'Donnell, [redacted], graduated from the United States Air Force Academy in May of 1997. His first assignment was to the Air Force Institute of Technology. Following graduation from AFIT, Lieutenant O'Donnell will be assigned to the 552nd Computer Systems Group at Tinker AFB, OK. He will be working with the software engineering process improvement flight.

PII Redacted

[redacted]

[redacted]

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
March 1999

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE
A Constructive Induction Approach to Computer Immunology

5. FUNDING NUMBERS

6. AUTHOR(S)
Capt Kelley J. Cardinale
Lt Hugh M. O'Donnell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Department of Electrical and Computer Engineering
2950 P Street
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/99M-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Capt Freeman Kilpatrick
AFOSR/NM
801 North Randoplh Street Room 732
Arlington, VA 22203-1977
(703) 696-6565

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Advisor: Lt Col Gregg Gunsch
(937) 255 -6565 x4281 (DSN) 785-6565 x4281
Gregg.Gunsch@afit.af.mil

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

With the increasing birth rate of new viruses and the rise in interconnectivity and interoperability among computers, the burden of detecting and destroying computer viruses is severe. This research integrated four domains: computer virus detection, human immunology, computer immunology and an automated form of machine learning called constructive induction. First, a Computer Health System, based on the public health system, was defined to improve the "global" approach to computer virus protection. Second, a computer immune model, based on the human immune system, was defined to improve the "local" approach to virus detection. Third, the detection component of this computer immune model was developed, represented by the prototype MERCURY. This model utilized constructive induction, capturing the human immune characteristics of detection, self-adaptation and memory.

The results of analyzing MERCURY demonstrate a lack of representational power of computer virus byte patterns using selective induction. Therefore, constructive induction is needed to provide new, potentially powerful, and often necessary representations. However, the results confirmed constructive induction's main deficiency, the explosion in the number of hypotheses generated. The effects of this deficiency can be improved by utilizing key pieces of knowledge to guide construction. Process optimization through statistical techniques provides insight into this knowledge.

14. SUBJECT TERMS
virus detection, antivirus, constructive induction, machine learning, classification, public health system, computer immunology, response surface methodology, process optimization

15. NUMBER OF PAGES
242

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassifed	Unclassifed	Unclassifed	